RTTI: Crie um framework de persistência e mapeamento objeto/relacional

Clube Delphi +PHP

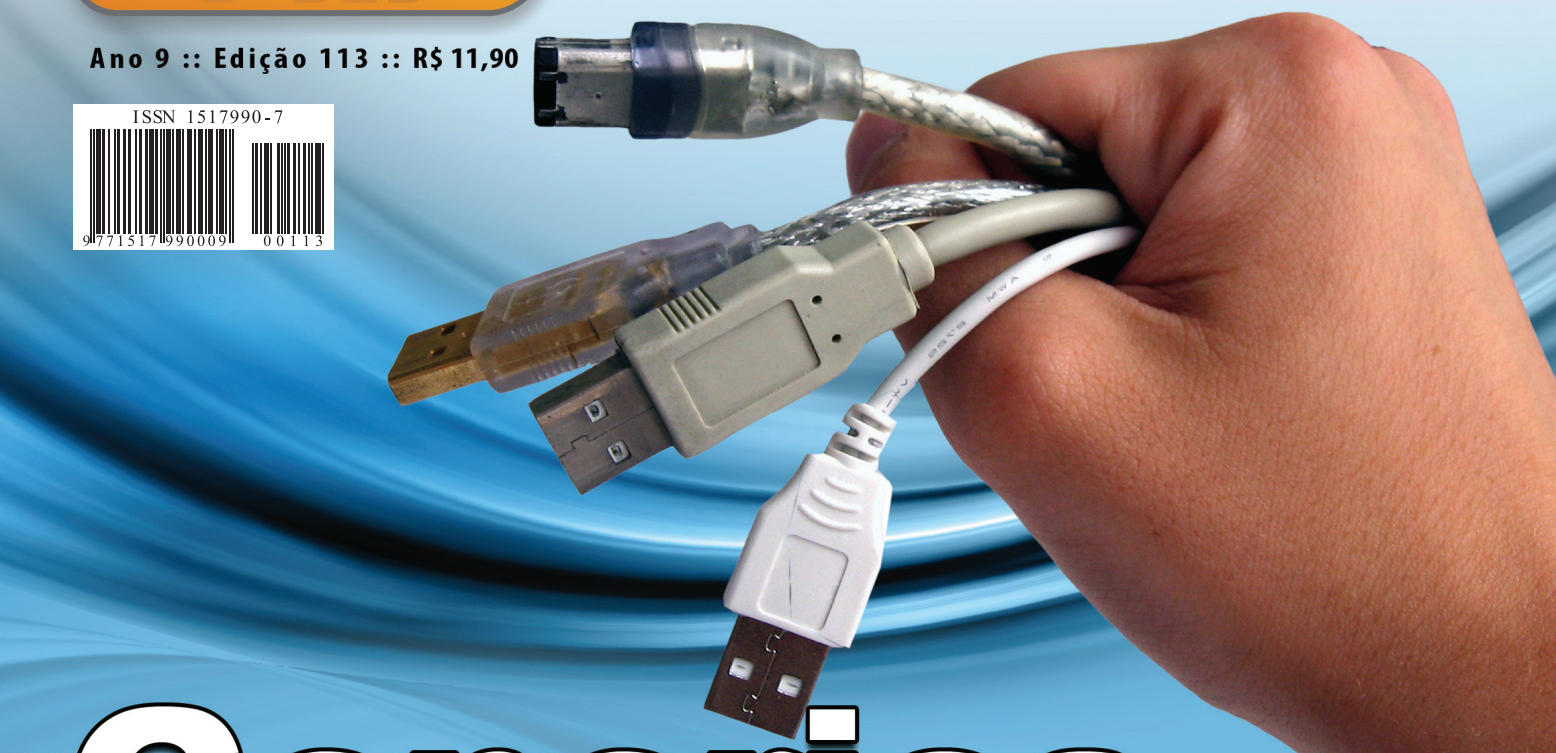
DevMedia group

Boas Práticas e Design Patterns
Encapsulamento ao extremo com o padrão Command

Easy Delphi
Curso de POO - Interfaces, abstração e polimorfismo em uma aplicação real

Ano 9 :: Edição 113 :: R$ 11,90

ISSN 1517990-7



Generics e RTTI

Brinde

Confira no portal ClubeDelphi PLUS 3 vídeo-aulas:

- Delphi Prism – eventos e delegates
- Atualização automática de sistemas
- Melhorias no debugger do Delphi 2010

local
Data Center

Nossos **servidores** dedicados possuem monitoramento 24x7x365, link dedicado e **full duplex**, ou seja, os servidores atingem a **velocidade** do link tanto para **upload** quanto para **download.**

HOST

Servidor
Sites
Gerenciamento

DATACENTER

Co-location
E-mail
Backup

TELECOM

Cross Connection
Banda IP
Link PAP

SOLUÇÕES INTELIGENTES

Solicite um representante comercial

www.localdatacenter.com.br

SOLICITE UM ORÇAMENTO
TEL.: 3956 - 4900
comercial@localdatacenter.com.br

Sumário

Delphi — Boa Ideia

5 – RTTI no dia–a–dia
Persistência de dados e formulários personalizados pelo usuário
[*Vitor Luiz Rubio*]

Boas Práticas — Boas Práticas, Engenharia de Software

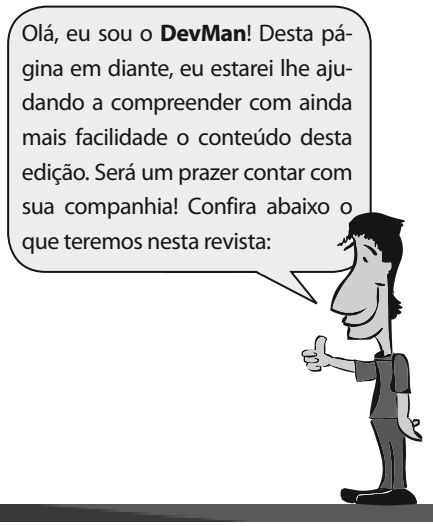
25 – Padrão Command
O Senhor Encapsulamento
[*Rafael Stavarengo*]

Boas Práticas — Boas Práticas, Boa Ideia

36 – Generics, RTTI e Custom Attributes
Desenvolva um mini framework de validação
[*Paulo Quicoli*]

Easy Delphi — Easy Delphi, Boas Práticas, Tutorial

46 – Introdução à POO – Parte 6
Aprenda de uma forma simples a programar da forma correta
[*Guinther Pauli*]



Brinde na web desta edição - 3 vídeos! Confira:

3 Vídeos

Delphi Prism - Eventos e Delegates no .NET Framework
http://www.devmedia.com.br/articles/viewcomp.asp?comp=14910

Atualizador Automático - Parte 1
http://www.devmedia.com.br/articles/viewcomp.asp?comp=14900

O novo Debugger do Delphi 2010
http://www.devmedia.com.br/articles/viewcomp.asp?comp=14901

Para acessar os vídeos utilize os seguintes dados
Login: ***DVM.PLS*** *Senha:* ***CDOOBP***

Gostou das vídeo aulas? O portal *www.devmedia.com.br* possui mais de **2 mil vídeo aulas** e **dezenas de cursos online** sobre desenvolvimento de software! Agora você pode comprar as vídeo aulas que preferir e fazer sua própria combinação de vídeos! Saiba mais em *www.devmedia.com.br/creditos*

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

Para votar, você vai precisar do código e senha de banca desta edição, que é:
Login: **DVM.PLS** Senha: **CDOOBP**

Clube Delphi +PHP

Ano 9 - 113ª Edição 2009 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral
Guinther Pauli
guinther@devmedia.com.br

Editor Técnico
Paulo Quicoli
pauloquicoli@devmedia.com.br

Daniel Sobrinho
daniel.sobrinho@devmedia.com.br

Rodrigo Carreiro Mourão
rodrigocarreiro@rmfactory.com.br

Capa
Romulo Araujo
romulo@devmedia.com.br

Diagramação
Janete Feitosa

Coordenação Geral
Daniella Costa - daniella@devmedia.com.br

Revisão e Supervisão
Thiago Vincenzo - thiago.v.ciancio@devmedia.com.br

Distribuição
Fernando Chinaglia Dist. S/A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

Cristiany Queiroz – Atendimento ao Leitor
www.devmedia.com.br/mancad
(21) 3382-5038

Kaline Dolabella – Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

Kaline Dolabella
publicidade@devmedia.com.br

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Guinther Pauli - Editor da Revista
guinther@devmedia.com.br

EDITORIAL

Boas Práticas. Este é o tema mais abundante desta edição, presente na maioria dos artigos. Nosso principal destaque vai para Generics, RTTI – Runtime Type Information e Custom Attributes no Delphi 2010, na matéria de Paulo Quicoli. Generics é um recurso muito, mas muito poderoso que já estava incorporado nas linguagens para .NET desde o .NET Framework 2.0, como C#, VB.NET, agora o Delphi Prism, o antigo Delphi for .NET e também em outras linguagens OO. Desde o Delphi 2009, os desenvolvedores Delphi puderam contar com este excelente recurso na linguagem Delphi, só que na plataforma Win32! Confesso que a primeira vista é muito estranho você se deparar com estruturas como TMeuArray<T> = array of T, ou ainda procedure Troca<T>(var Obj1,Obj2:T). Aliado ao poder dos Generics, temos a nova RTTI do Delphi 2010 e suporte a Atributos em classes, propriedades, métodos etc. Enfim, temos aqui três belos ingredientes que podem dar às suas aplicações maior flexibilidade e performance, com muito reaproveitamento de código. E nada melhor do que aplicar todas as técnicas em um exemplo real, como de costume por aqui.

RTTI também é o tema principal do artigo do Vitor, que começa mostrando o básico sobre RTTI ("is","as") e aprofunda até criar um mini framework de mapeamento objeto/relacional, para persistência de objetos em um servidor SQL. Essa é uma das principais aplicações para a RTTI, e se você adicionar aí na receita os ingredientes do artigo do Paulo, poderá criar frameworks extremamente robustos. Excelente prática.

Mais boas práticas! Encapsulamento, um dos pilares da Orientação a Objetos. Um simples Get / Set escondendo um atributo em uma classe já é considerado uma forma de encapsulamento. Indo mais além, entrando na área de Design Patterns, o padrão Command é usado para encapsular a chamada de um método. O Rafael traz nesta edição um belíssimo artigo sobre este padrão, mesclando ótimas práticas e muita OO, como Interfaces. Você aprenderá a criar um sistema fechado contra erros e ao mesmo tempo aberto para expansão, graças ao encapsulamento.

A nossa sessão Easy também não foge do contexto principal desta edição! Mesmo os principiantes no Delphi podem e devem aplicar boas técnicas em suas aplicações. Como costumo dizer, a OO e os Padrões de Projeto se resumem em basicamente duas regras: Separar e Esconder. Finalizando nosso mini-curso, mostrarei como essas duas práticas simples podem fazer a diferença na hora de resolver um problema em um sistema de nível altamente complexo e crítico (aqueles que não se pode errar de forma alguma, como um sistema de vestibular, o caso real mostrado aqui).

Boa leitura. E boas práticas, claro!

Guinther Pauli
guinther@devmedia.com.br
Certificado Delphi 3,5,6,7,2005,2006,Web,Kylix
MCP,MCAD,MCSD.NET,MCTS,MCPD
guintherpauli.blogspot.com
twitter.com/guintherpauli

ClubeDelphi PLUS

DevMedia group

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
http://www.devmedia.com.br/clubedelphi/portal.asp

Nesta seção você encontra artigos intermediários sobre Delphi Win 32 e Delphi for .NET

RTTI no dia–a–dia

Persistência de dados e formulários personalizados pelo usuário

Do que trata o artigo

Reflexão ou introspecção é a capacidade que muitas linguagens de programação modernas têm de permitir que se criem classes com o conhecimento de sua própria estrutura. Quando as classes "sabem" como são feitas, você pode obter valores de propriedades através de strings, sem necessariamente ter certeza de que a classe em questão realmente tem tal propriedade. Faremos isso usando o mecanismo de RTTI (Runtime Type Information – Informação em tempo de execução) do Delphi, que nos permite obter informações sobre a estrutura dos objetos instanciados. Mas não exploraremos apenas a RTTI, mas mostraremos como ela pode ser utilizada na flexibilização de projetos orientados a objetos.

Para que serve

Perguntando para um objeto se ele tem determinada propriedade e depois obtendo o seu valor nos permite criar em nossas classes métodos genéricos, que podem trabalhar literalmente com qualquer tipo de parâmetro.

Em que situação o tema é útil

Essa técnica pode ser muito útil quando o programador tem um objeto instanciado do tipo TForm, mas não sabe exatamente que form é e precisa consultar ou modificar o valor de determinado atributo.

Resumo do DevMan

Serão expostas algumas técnicas de RTTI para se obter informações de objetos e persistir essas informações em banco de dados. Serão mostrados desde a criação das tabelas aos métodos para inserir, alterar e excluir registros. Também será apresentado como podemos criar interfaces que o próprio usuário possa alterar, diminuindo assim os custos de manutenções simples como trocar a cor ou posição de controles no formulário.

Trabalhando com desenvolvimento de software nos deparamos com inúmeras situações em que nossos clientes nos pedem customizações. Algumas vezes estas são mudanças significativas, mas interessantes, que podemos incorporar em nossos produtos e agregar valor aos nossos projetos. Entretanto a maioria das vezes essas alterações são úteis para apenas um cliente, às vezes nos forçando a trabalhar com dois projetos em paralelo.

Vitor Luiz Rubio
vitorrubio@gmail.com
Analista de Sistemas Sr. na Editora Revista dos Tribunais. Trabalha com Delphi desde a versão 3. Formado em Processamento de Dados pela FATEC-SP.

Essa tendência nos leva a criar sistemas bastante customizáveis, com muitos parâmetros de configuração. Estamos nos habituando a programar orientado a objetos criando classes cada vez mais genéricas para que possamos trocar livremente os objetos, unidades e componentes do nosso sistema.

Quando precisamos passar como parâmetro para um método de uma classe um objeto de uma classe qualquer já instanciado, numa variável do tipo TObject, para ser "genérico", dentro do escopo do método precisamos saber de que tipo esse objeto é, para podermos alterar seguramente as suas propriedades e invocar suas funções e procedimentos. Há uma maneira mais elegante de tratar esses objetos genéricos do que fazer Typecasts, muitas vezes inseguros, ou uma lista interminável de desvios condicionais com muitos if's e cases.

As linguagens de programação modernas fornecem um recurso chamado Introspecção, ou Reflexão, que é a habilidade de uma classe de "conhecer" detalhes sobre sua estrutura, seus metadados, métodos e propriedades.

O Delphi provê essa habilidade na sua RTTI (Runtime Type Information - Informação em Tempo de Execução). Criada há mais de 10 anos, a RTTI tinha o objetivo inicial de facilitar o desenvolvimento da própria IDE do Delphi, para que ela pudesse cumprir com uma de suas missões que é "conhecer" as classes que estão sendo criadas ou usadas pelo desenvolvedor. Ela é amplamente utilizada na nossa ferramenta, por isso que o Object Inspector sabe de cada propriedade, valor e evento do componente que selecionamos, não importando qual componente seja.

Embora a RTTI nunca tenha sido bem documentada fora da equipe de desenvolvimento do Delphi, ela sempre pôde ser usada por programadores para facilitar algumas de suas tarefas na programação orientada a objetos.

Vamos ver como essa tecnologia pode nos ajudar a economizar bastante tempo, criar interfaces customizáveis para nossos clientes e usar a RTTI no dia-a-dia.

Conhecendo o básico

Talvez o leitor já use RTTI e não sabe. Sempre que usamos os operadores "is" para perguntar se um objeto é de um determinado tipo e "as" para fazer um Typecast seguro estamos usando na verdade RTTI.

Vamos ao código, que fala muito melhor a nossa língua. Usaremos neste exemplo o Turbo Delphi 2006, que pode ser obtido pelo site da Embarcadero. Fique à vontade para usar versões mais atuais do Delphi, apenas fique atento ao usar a versão 7 ou 2010. A versão 7 possui algumas diferenças, e o a 2010 teve a RTTI bastante modificada, assunto que já foi abordado aqui na revista e não focaremos neste artigo.

Crie uma nova aplicação Delphi VCL Forms para win32, renomeie o Form1 para frmTesteRTTI. Esta será nossa form principal e todos os nossos testes e exemplos partirão dela. Salve o projeto como ExemploRTTI.dpr e a Unit1 como uFrmTesteRTTI.pas. Inclua um Edit com o nome de edTeste e um Button com o nome de btTesteEdit.

Crie uma procedure chamada AlteraTexto, que receba 2 parâmetros, um TObject e uma string. Essa procedure será responsável por mudar o texto de nosso Edit e de qualquer outro componente que colocarmos no formulário. E depois vamos codificar o evento onClick do botão btTesteEdit para chamar esse procedimento passando como primeiro parâmetro o edTeste e como segundo parâmetro a mensagem super original "Alô Universo!".

Neste método usaremos os operadores "is" para saber o tipo do objeto passado como parâmetro e o operador "as" para converter para o tipo que precisamos. Esse código pode ser visto na **Listagem 1.**

Listagem 1. Procedure AlteraTexto e botão btTesteEdit

```
//método público deste form: AlteraTexto
procedure TfrmTesteRTTI.AlteraTexto(obj: TObject; texto:
string);
begin
  if obj is TEdit then     //verifica: obj é mesmo um edit?
  (obj as TEdit).Text := texto; //então, sendo um edit,
  modificamos a propriedade Text
end;

procedure TfrmTesteRTTI.btTesteEditClick(Sender: TObject);
begin
  AlteraTexto(edTeste, 'Olá Universo!');
end;
```

Este código não faz nada mais que usar o operador "is" para saber de que classe que é o objeto e o operador "as" para fazer typecast seguro e acessar suas propriedades. Uma das várias vantagens do Typecast seguro é que, se você tentar converter para o tipo errado, a instrução não vai passar deste ponto e causar um access violation depois da conversão, mas vai disparar uma exceção do tipo EinvalidCast com a mensagem "Invalid class typecast" no momento da conversão.

Até agora, nenhuma novidade. Esta procedure que criamos não é muito útil, mas repare que ela não necessariamente precisaria estar neste formulário. Ela poderia estar em qualquer uma das nossas classes de negócio, servindo inclusive para preencher nossos forms.

Vamos criar um outro botão, com o nome de btTesteCombo e adicionar um ComboBox com o nome cmbTeste. Nosso formulário ficará como a **Figura 1.**

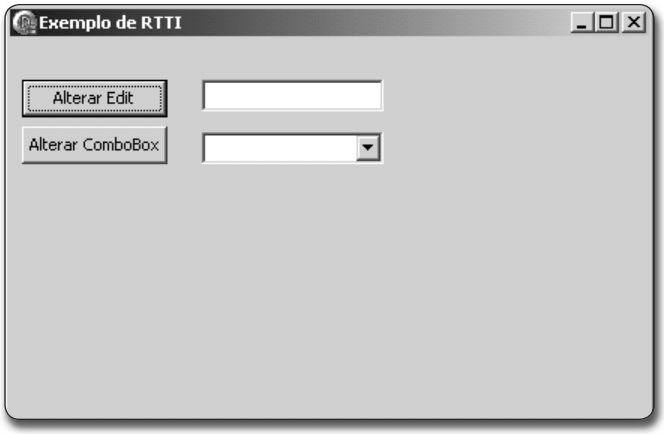


Figura 1. Form com Edit, ComboBox e botões

Vamos fazer a mesma coisa agora para verificar o ComboBox. No código do btTesteCombo, repetimos o mesmo código do botão anterior, apenas passando o cmbTeste como primeiro parâmetro. A procedure AlteraTexto deve ser mudada para suportar o ComboBox, adicionando um if para verificar se o tipo é um TEdit ou um TComboBox, conforme a **Listagem 2.**

Listagem 2. Procedure AlteraTexto com ComboBox

```
procedure TfrmTesteRTTI.AlteraTexto(obj: TObject; texto:
string);
begin
  if obj is TEdit then
    (obj as TEdit).Text := texto
  else
  if obj is TComboBox then
    (obj as TComboBox).Text := texto
end;

procedure TfrmTesteRTTI.btTesteEditClick(Sender: TObject);
begin
  AlteraTexto(edTeste, 'Olá Universo!');
end;

procedure TfrmTesteRTTI.btTesteComboClick(Sender: TObject);
begin
  AlteraTexto(cmbTeste, 'Olá Universo!');
end;
```

O problema que nós criamos: além de nosso código precisar de inúmeros if's, um para cada tipo de componente, criamos a inconveniência de que a unit que tiver o nosso método AlteraTexto deve conhecer todas as classes usadas: TEdit, TComboBox e outras que possam ser adicionadas. Ela deverá ter várias units da VCL no seu uses. Isso não é bom se quisermos, no futuro, trocar todos os nossos componentes visuais por componentes da biblioteca JEDI ou da DevExpress, por exemplo (ou mesmo Web).

Nota

Embora ComboBox e Edit tenham algumas propriedades em comum, seu grau de parentesco é bem distante, somente em TWinControl eles encontram um ancestral em comum, sendo em TControl, pai de TWinControl, que eles herdam a propriedade Text. Mesmo existindo esse parentesco, você pode considerar ComboBox e Edit como duas classes totalmente distintas, embora o typecast para TWinControl também funcionasse. Esse exemplo é apenas ilustrativo, e serve para demonstrar o que fazer com classes sem nenhuma relação.

A unit TypInfo

A unit TypInfo contém tipos de dados e métodos especiais para se trabalhar com informações de tipos, tanto em tempo de design como em tempo de execução, em vários níveis diferentes. Foge ao escopo desse artigo detalhar ou documentar completamente o que essa unit possui, mas alguns conceitos são necessários ao que nos dispomos a fazer.

Vamos alterar a nossa procedure AlteraValor para não fazer nenhuma referência a nenhuma classe, não fazer typecast e simplesmente obter ou mudar o valor de uma propriedade pelo seu nome. Para isso, não esqueça de colocar TypInfo na seção uses do form. Vamos criar uma sessão "var" na procedure AlteraValor e criar uma variável chamada InfoPropriedades do tipo PPropInfo. Este tipo é um ponteiro para TPropInfo, que é uma estrutura de dados que armazena uma tabela de informações sobre propriedades publicadas de qualquer tipo de classe que seja derivada de TPersistent. Vamos atribuir a InfoPropriedades o resultado da função GetPropInfo, passando como parâmetros o objeto e o nome da propriedade, no nosso caso, "Text".

Essa função GetPropInfo possui vários overloads. Isso te dá a liberdade de fazer a mesma coisa de várias formas. Experimente tentar fazer a mesma coisa, mas passando a classe do objeto ao invés de sua instância.

O que vamos fazer aqui será atribuir um valor a uma propriedade de um objeto através do nome da propriedade. Isso mesmo, o nome como uma string. Iremos verificar: "O objeto tem a propriedade X? Se tiver então atribua a propriedade X o valor Y". Seu código ficará como a **Listagem 3.**

Listagem 3. Usando funções da TypInfo

```
procedure TfrmTesteRTTI.AlteraTexto(obj: TObject; texto: string);
var
  //PPropInfo é um ponteiro para uma tabela de informações sobre
   propriedades
  infoPropriedades: PPropInfo;
begin
    //usa um objeto e o nome da propriedade para obter um
    ponteiro para as informações desta propriedade
    infoPropriedades := GetPropInfo(obj, 'Text');
     if Assigned(infoPropriedades) then
            //atribui um valor a uma propriedade através de seu nome.
             SetStrProp(obj, 'Text', texto);
end;
```

O que fizemos foi usar o método GetPropInfo(obj, 'Text') para obter um ponteiro para uma estrutura de dados chamada TPropInfo que contém informações sobre a propriedade. No caso pesquisamos a propriedade Text do objeto. Se o objeto tiver uma propriedade chamada "Text" o método retornará um ponteiro válido, mas se a propriedade "Text" não existir nesse objeto, então o método retornará um ponteiro nulo (nil). Por isso verificamos se infoPropriedades é um ponteiro que aponta realmente para algum lugar, usando o método Assigned(infoPropriedades). Se o resultado for verdadeiro, podemos atribuir qualquer valor à propriedade "Text" através do método SetStrProp(obj, 'Text', <texto a ser atribuido>).

Esse método é usado para strings, caso a propriedade seja uma string, mas existem métodos similares para trabalhar com integers, enums, variants etc.

SetStrProp(obj, 'Text', 'um texto'), por exemplo, seta o valor 'um texto' à propriedade Text do objeto obj, se ele possuir uma propriedade Text. Por isso verificamos antes, caso contrário será disparada uma exceção.

Isso abre um leque de novas possibilidades. Podemos atribuir qualquer valor a qualquer propriedade de qualquer objeto, bastando que ele tenha a propriedade. Podemos armazenar em bancos de dados pares de Propriedade x Valor para cada componente, de cada form, que desejamos permitir que o usuário configure, sem ter que recompilar o projeto para mudar essas propriedades.

Como pudemos ver, estamos usando ponteiros. Ponteiros são uma parte fundamental da programação, e estudá-los é altamente recomendável. Muitos programadores nunca ouviram falar de ponteiros porque sempre programaram em

linguagens interpretadas e/ou gerenciadas, como Java, C#, PHP, Python, Ruby, VB e assim por diante. Nessas linguagens o uso e gerenciamento de memória e ponteiros é transparente. Você pode entender ponteiros como um endereço de memória ou como uma referência para um determinado local da memória. Esse local pode ser a área da memória onde você declarou uma variável, mas também pode apontar para uma função ou vetor.

Se tudo que declaramos é armazenado em algum lugar da memória e esses lugares são "numerados", uma variável do tipo ponteiro é uma variável que tem um valor, geralmente um inteiro de 4 bytes, que aponta para o endereço de memória de alguma coisa. No nosso caso a variável infoPropriedades vai ter o endereço de uma estrutura de dados que contém informações em tempo de execução de uma propriedade, por exemplo a propriedade Text do Edit.

Embora o uso de ponteiros possa parecer programação de baixo nível, não é tão baixo assim, pelo menos não como assembly. Digamos que é programação de nível intermediário. Estruturas de dados complexas e encadeadas usam ponteiros. E a API do Windows usa muito ponteiros, estruturas, ponteiros para estruturas e assim por diante. Você pode ver esses usos dando uma olhada nas units da VCL e da RTL. Aprende-se muito com essas units.

Não vamos nos aprofundar em ponteiros porque eles dariam assunto para muitos artigos, e uma boa seção de um livro.

Listando todas as propriedades e valores

O estado de um objeto pode ser definido como os valores de seus atributos. Atributos são valores, podem ser campos ou propriedades, públicos ou privados. Lógico que uma parte dos atributos são campos privados, e outra parte são mudados pelas propriedades através dos métodos Get e Set, mas se uma classe possuir todos os seus atributos relevantes como propriedades públicas e puder reconstituir seus campos privados através de propriedades e métodos, é bem possível que a interface dessa classe, ou seja, suas propriedades e métodos públicos já seja suficiente para expressar o estado do objeto.

Para uma classe gerar informações para a RTTI ela deve ser descendente de TPersistent ou deve ter a diretiva de compilação {$M+} ativada no início do escopo da classe e desativada {$M-} no final (lembre-se que houve mudanças no Delphi 2010). Essa diretiva faz com que a classe armazene informações sobre as propriedades e métodos.

No artigo vamos usar sempre TPersistent, e TInterfacedPersistent quando quisermos trabalhar com interfaces, porque essas duas classes já ativam automaticamente a diretiva {$M}, conforme você pode observar na **Listagem 4.**

Este trecho de código da **Listagem 4** foi extraído da unit Classes do Delphi. É bom mencionar que é justamente por esse motivo que nossos exemplos funcionaram até agora. Fizemos os exemplos com Edit e ComboBox, e todos os componentes são, lá no fundo, descendentes de TPersistent. Criar suas classes como filhas de TPersistent é a maneira mais fácil de assegurar que elas terão suporte a RTTI.

Nota do DevMan

Sempre que você usa um objeto, está usando na realidade um ponteiro. Pode verificar isso pelo tamanho das variáveis de um tipo objeto, que todas tem 4 bytes. O objeto propriamente dito está alocado em algum lugar da memória, que é referenciado pela variável que aponta para ele. Ponteiros também são muito úteis na implementação de estruturas de dados como pilhas, filas, listas ligadas e árvores binárias.

Um uso de ponteiros transparente é quando passamos um argumento para uma função por referência. Passar um argumento por referência significa literalmente que estamos passando um ponteiro para a variável, e por isso que a própria será alterada. Podemos simular a passagem de um parâmetro por referência passando um ponteiro para uma variável. O operador de ponteiro ^ usado ao se declarar uma variável ou tipo, com a sintaxe i: ^integer, é usado para declarar uma variável ou tipo ponteiro. O mesmo operador usado depois de um identificador, por exemplo i^ := 2 significa o valor armazenado no local da memória para onde o ponteiro aponta. O operador @ obtém um endereço de memória de uma variável, por exemplo p := @x.

Listagem 4. Declaração de TPersistent

```
{$M+}

  TPersistent = class(TObject)
  private
    procedure AssignError(Source: TPersistent);
  protected
    procedure AssignTo(Dest: TPersistent); virtual;
    procedure DefineProperties(Filer: TFiler); virtual;
    function  GetOwner: TPersistent; dynamic;
  public
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); virtual;
    function  GetNamePath: string; dynamic;
  end;

{$M-}
```

Vamos criar a nossa própria classe de clientes para mostrar como podemos listar todas as propriedades e valores de um objeto instanciado. Com tamanho poder de fogo temos muitas possibilidades. Por exemplo, copiar um objeto: o código da **Listagem 5** não copia um objeto, copia uma referência (endereço ou ponteiro) para um objeto, mas as duas variáveis ainda apontam para o mesmo local na memória. O que você mudar em uma, muda na outra, porque na verdade são o mesmo objeto. Se você destruir uma e tentar executar um método da outra terá um erro.

Listagem 5. Como NÃO copiar um objeto

```
var
  a, b: TStringList;
begin
//esse código não copia o objeto, mas copia a refêrencia.
(ponha classes no uses)
//...
  a := TStringList.Create;
  a.Add('a');
  b := a;  //isto copia a referência e não o objeto
  b.Add('b');
  showmessage(b.Text);
  a.Free;
  b.Add('c');
  a.Add('d');
  ShowMessage(a.Text);
  ShowMessage(b.Text);
//...
```

O maior perigo do código da **Listagem 5** é fazermos algo parecido por distração ou falta de atenção. Depois que executamos o método Free de uma das variáveis, a outra passa a apontar para uma referência inválida. Aí nós teremos na melhor das hipóteses uma exceção do tipo "Access Violation".

Geralmente usamos o método Assign para copiar um objeto para outro, então é obrigatório que este método esteja implementado. Dificilmente nos lembramos de implementar o método Assign em nossas próprias classes. Com a técnica de listar propriedades e valores via RTTI você pode criar um método que gere um clone perfeito de um objeto qualquer. E pode inclusive usar isso ao implementar os seus métodos assign, ao invés de ter o trabalho chato de: obj1.propriedade := obj2.propriedade. Veremos como criar esse método de copiar objetos mais a frente.

Um framework de persistência usando RTTI

Agora nos preocuparemos com persistência em banco de dados. Vamos um pouco mais além, o que vamos fazer aqui é usar RTTI para criar os metadados de uma tabela no banco de dados, caso ela não exista, e inserir nesta tabela um registro baseado no estado do objeto, uma coluna da tabela por propriedade. Depois veremos como carregar um objeto com informações do banco, executar um método pelo seu nome e persistir em banco de dados informações sobre a configuração da interface com o usuário.

Esse nosso exemplo está longe de ser um framework de persistência de objetos, como um Hibernate, NHibernate ou ADO.NET Entity Framework, por exemplo, e existem vários frameworks prontos para isso. Mas é a base para você criar seu próprio e adquirir conhecimento. Estude-o e refine-o. Também aqui não daremos demasiada importância ao banco de dados, ou aos melhores métodos para se gravar ou ler dados, porque não é nosso objetivo. Além disso, nessa área de banco de dados, sempre houve uma boa discussão entre os fanáticos por "tudo com procedures" e os partidários de "nada com procedures", por isso eu deixo o SQL por sua conta.

Focando no nosso código, vamos começar pela classe de clientes. Você pode usá-la como base na hora de criar outras classes de "Entidades", como fornecedores, funcionários, contatos, produtos etc. Criaremos a classe mais simples possível. Nossa classe será descendente de TPersistent e terá propriedades Nome, DataNascimento, CPF, e ID. Crie esta classe em uma unit separada, dando-lhe o nome de uCliente.pas.

Nota

A classe TPersistent fica na unit classes do Delphi, por isso não esqueça de colocá-la no uses da uCliente.

O código da **Listagem 6** mostra como será nossa classe. Apesar de apenas lermos e gravarmos os valores das propriedades direto nos campos privados correspondentes, é uma boa prática utilizar os métodos acessores Get e Set para qualquer alteração futura que envolva validação ou obtenção indireta desses atributos.

Listagem 6. Classe TCliente

```
unit uCliente;

interface

uses Classes;

type
  TCliente = class(TPersistent)
  private
    FID: integer;
    FNome: string;
    FCPF: string;
    FDataNascimento: TDateTime;
  protected
    //gets
    function GetID: integer;  virtual;
    function GetNome: string; virtual;
    function GetCPF: string;  virtual;
    function GetDataNascimento: TDateTime; virtual;
    //sets
    procedure SetID(const valor: integer);   virtual;
    procedure SetNome(const valor: string); virtual;
    procedure SetCPF(const valor: string);  virtual;
    procedure SetDataNascimento(const valor: TDateTime);  virtual;
  published
    property ID: integer read GetID write SetID;
    property Nome: string read GetNome write SetNome;
    property CPF: string read GetCPF write SetCPF;
    property DataNascimento: TDateTime read GetDataNascimento
     write SetDataNascimento;
  end;

implementation

{ TCliente }

function TCliente.GetCPF: string;
begin
  Result := FCPF;
end;

function TCliente.GetDataNascimento: TDateTime;
begin
  Result :=  FDataNascimento;
end;

function TCliente.GetID: integer;
begin
  Result :=   FID;
end;

function TCliente.GetNome: string;
begin
  Result :=  FNome;
end;

procedure TCliente.SetCPF(const valor: string);
begin
  FCPF := valor;
end;

procedure TCliente.SetDataNascimento(const valor: TDateTime);
begin
  FDataNascimento := valor;
end;

procedure TCliente.SetID(const valor: integer);
begin
  FID := valor;
end;

procedure TCliente.SetNome(const valor: string);
begin
  FNome := valor;
end;

end.
```

Adicione no form mais um botão, com o nome btVerPropriedades e um Memo com o nome mTexto, deixando-o como a **Figura 2.**

Não vamos nos limitar somente a listar o nome e valor da propriedade, mas também mostrar o seu tipo. Isso será útil para a formatação da saída. Por exemplo, todos nós sabemos que o tipo TDateTime é no fundo um double. Para não apresentarmos

uma propriedade do tipo data de forma numérica, iremos antes verificar se o "nome do tipo de dado" da referida propriedade é TDateTime.

Figura 2. Form com botão e Memo para listar propriedades de TCliente

A unit TypInfo contém vários métodos para obter ou setar valores de propriedades. Nós usaremos neste exemplo o GetPropValue que retorna um variant. É suficiente na maioria dos casos. Mas você pode estudar e usar os outros métodos ou até criar os seus próprios quando estiver lidando com datas, enums, sets, arrays e outros tipos.

Para a correta formatação da nossa saída vamos criar um método onde passaremos como parâmetros um objeto e um TPropInfo. O método verificará o tipo e o nome do tipo da propriedade e formatará a saída de acordo com a necessidade. Para isso usaremos o atributo PropType do TPropInfo que é um ponteiro para uma estrutura de dados (Record) que armazena informações sobre tipo. Como esse atributo é um ponteiro teremos de usar o operador ^ logo após o seu nome para acessar os seus membros. O membro name contém o nome do tipo da propriedade, enquanto o membro kind é um enum que corresponde ao tipo primitivo verdadeiro da propriedade. Por exemplo, uma propriedade do tipo TDateTime, o nome do tipo é "TDateTime", mas o kind é tkFloat. Podemos ver esse método na **Listagem 7.**

Listagem 7. Informações sobre a propriedade, formatadas

```
function TfrmTesteRTTI.FormataPropriedade(Instancia: TObject;
 const PropInfo: TPropInfo): string;
begin
  if (PropInfo.PropType^.Kind = tkFloat) and
    (PropInfo. PropType^.Name = 'TDate') then
    Result := FormatDateTime('yyyy-mm-dd', GetPropValue
    (Instancia, PropInfo.Name))
  else  if (PropInfo.PropType^.Kind = tkFloat) and
    (PropInfo.PropType^.Name = 'TTime') then
    Result := FormatDateTime('hh:nn:ss', GetPropValue(Instancia,
    PropInfo.Name))
  else if (PropInfo.PropType^.Kind = tkFloat) and
    (PropInfo.PropType^.Name = 'TDateTime') then
    Result := FormatDateTime('yyyy-mm-dd hh:nn:ss', GetPropValue
    (Instancia, PropInfo.Name))
  else
    Result := VarToStr( GetPropValue(Instancia, PropInfo.Name));
end;
```

Explicando o código da **Listagem 7**. Quando o tipo de dado é data, hora ou data e hora, precisamos de uma formatação especial para exibir como texto. Caso contrário convertemos de variant para string normalmente. Para formatar a exibição dos outros tipos de dados basta ampliar este método.

O ponto mais interessante fica por conta do membro PropType de PropInfo. A estrutura TPropInfo possui um membro chamado PropType que é do tipo PPTypeInfo, que é um "ponteiro para outro ponteiro", que é um record com informações sobre o tipo de um dado, com o nome do tipo e o seu "TypeKind". Se o tipo TPropInfo tem informações sobre uma propriedade, o seu TTypeInfo tem informações sobre o tipo desta propriedade, seja ela string, integer, float etc.

Coloque uCliente no uses da nossa form principal, a frmTesteRTTI, e no evento OnClick do botão btVerPropriedades, vamos criar uma variável do tipo TCliente e preencher algumas propriedades. Usaremos a função GetPropList que aceita como parâmetro um objeto instanciado e uma variável do tipo PPropList que é um ponteiro para TPropList.

Para manipularmos o operador de ponteiro ^ o mínimo possível criaremos uma variável do tipo PPropList chamada plistaProps para passar como parâmetro para GetPropList, e uma variável listaProps, que não é ponteiro, para receber o valor de plistaProps com a linha de comando: listaProps := plistaProps^;

Nota

Não tente usar essa abordagem no Delphi 7 porque não funcionará. Como o tipo TPropList é derivado de um array de número fixo de elementos, o Delphi 7 não sabe como copiar esta variável para uma outra automaticamente, por isso você pode ou copiar manualmente as variáveis do tipo vetor com o auxílio de um método ou usar o operador de ponteiro ^ depois de todas as variáveis ou métodos do tipo PPropList. Logo depois de digitar o ^ você tem acesso aos membros de TPropList na memória.

Como o tipo TPropList é um array unidimensional fixo de várias posições do tipo PPropInfo, que por sua vez é um ponteiro para TPropInfo, a estrutura que contém as informações sobre as propriedades de um objeto, também criaremos uma variável chamada infoProps do tipo TPropInfo para receber o valor da posição corrente do array listaProps com a linha: infoProps := listaProps[i]^;

Nota

Não faça essa atribuição no Delphi 7, use direto o ponteiro listaProps[i]^ para acessar os membros que precisar.

Nunca esqueça de inicializar os ponteiros que você usar com nil. A maneira usual de se usar ponteiros é alocar memória para eles com GetMem e depois liberar a memória com freeMem.

Como a função GetPropList já chama GetMem internamente e aloca memória para nosso ponteiro, você não precisa se preocupar com a alocação da memória, mas deve se preocupar com a desalocação. Você não deve esquecer de desalocar essa memória com FreeMem, se não criará um memory leak. Também não deve alocar novamente memória para esse ponteiro com GetMem, senão criará uma área de memória nova e limpa, que perderá sua referência quando GetPropList chamar GetMem novamente, ficando impossível de desalocar essa memória, mesmo com FreeMem, criando outro memory leak.

Vamos agora fazer um pequeno teste. Vamos instanciar um objeto da classe TCliente e listar suas propriedades. Para isso usaremos o método GetPropList da unit TypInfo que retorna o

número de propriedades publicadas de um objeto e preenche um array com um ponteiro PPropInfo para cada propriedade. Varrendo esse array com um laço for você pode acessar cada elemento PPropInfo e assim obter as informações de cada uma das propriedades. Esse array será representado pela variável listaProps, do tipo TPropInfo. TPropInfo é um tipo derivado de array, conforme definido na unit TypInfo:

```
PPropList = ^TPropList;
TPropList = array[0..16379] of PPropInfo;
```

A **Listagem 8** contém o código do evento OnClick do botão btVerPropriedades. As explicações sobre as variáveis, estruturas e métodos utilizados estão nos comentários do próprio código. A forma correta de se usar o GetPropList e FreeMem você pode ver nesse exemplo. Também não esqueça de sempre destruir qualquer TCliente que você criar, seja para teste, seja para persistir no banco.

O resultado ao se clicar no botão pode ser visto na **Figura 3.** Tomei a liberdade de teclar uns "TABS" no memo apenas para alinhar as colunas e visualizar melhor os atributos das propriedades.

Listagem 8. Listando propriedades, tipos e valores

```
procedure TfrmTesteRTTI.btVerPropriedadesClick(Sender: TObject);
var
  obj: TCliente;
  infoProps: TPropInfo;   //informações sobre a propriedade
  plistaProps: PPropList; //ponteiro para array de ponteiros
   sobre informações da propriedade
  listaProps: TPropList;  //array de ponteiros para informações
   da propriedade
  i: integer;
  totalProps: integer;    //valor total de propriedades do objeto
begin

  //sempre inicialize o ponteiro como nil
  plistaProps := nil;

  //setando algumas propriedades do nosso objeto
  obj := TCliente.Create;
  obj.ID := 1;
  obj.Nome := 'Clube Delphi';
  obj.CPF := '11111111111';
  obj.DataNascimento := StrToDate('31/12/2000');

  try
    mTexto.Lines.Add(
      'Propriedade:'+#9#9#9+
      'Tipo:'+#9#9#9 +
      'Valor');

    //a função  GetPropList seta o ponteiro para lista de
    propriedades e também retorna o total de propriedades
    totalProps := GetPropList(obj, plistaProps);
    listaProps := plistaProps^;   //atribuindo a  listaProps o
    valor armazenado no endereço plistaProps
    for I := 0 to totalProps - 1 do
    begin
      infoProps := listaProps[i]^; //atribuindo a infoProps o
       valor contido no endereço da posição i do vetor listaProps

       //adiciona no memo o nome, tipo e valor da
       propriedade, separado por 3 tabs, formatado
       mTexto.Lines.Add(
       infoProps.Name+#9#9#9+
       infoProps.PropType^.Name+#9#9#9 +

               FormataPropriedade(obj, infoProps));
    end;
  finally
    obj.Free;
    //não podemos esquecer de liberar a memoria do ponteiro
    FreeMem(plistaProps);
  end;

end;
```

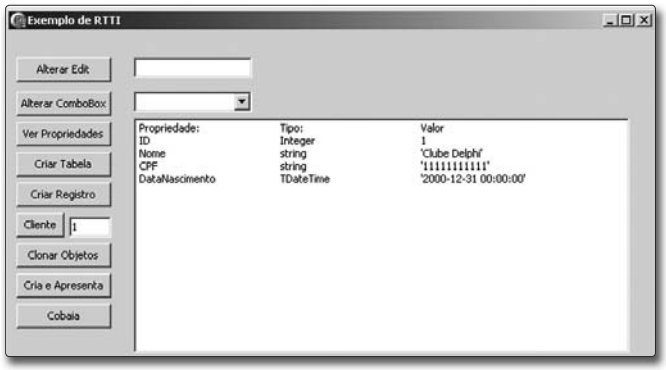


Figura 3. Clicando no botão "Ver Propriedades"

Nota

Não se assuste com a quantidade de botões no formulário da Figura 3. Nós criaremos todos eles.

Embora bem básico, isso que já temos serve como base para persistirmos esse objeto em banco de dados. Crie um banco vazio no sistema gerenciador de banco de dados que você desejar, Interbase, Firebird, SQL Server, Postgre etc. Apenas tome o cuidado de adaptar os seus comandos SQL de acordo com o banco de dados em questão. Neste exemplo estamos usando SQL Server 2005 e criamos o banco, chamado dbRTTI, através do SQL Server Management Studio.

Nota do Editor

Você pode baixar uma versão gratuita do SQL Server Express e também do gerenciador front-end (veja sessão links). Ambos são extremamente fáceis de instalar e usar. Após instalar o SQL Server Express, basta abrir o Management, logar, dar um clique de direita sobre o item Databases e criar o banco deste exemplo.

Crie um TDataModule na sua aplicação dando o nome de DmPersistencia e chamando a sua unit de uDmPersistencia. Você poderia também criar a classe de persistência e o datamodule em duas classes e units separadas, ou não usar datamodule e colocar os componentes de acesso a dados na form principal. Estamos usando o datamodule para fins de organização e didática. Coloque no seu datamodule um ADOConnection nominado adoConn e um ADOCommand chamado adoCom. Configure a connection string de seu AdoConn para acessar seu banco de dados, de acordo com o nome e endereço de seu banco, e configure a propriedade Connection de adoCom para adoConn.

Nota do Editor

Para configurar a conexão com o banco, dê um duplo clique no ADOConnection, escolhe o provider nativo para SQL Server, informe .\SQLEXPRESS no nome do servidor (no caso de estar usando a versão express), opte pela autenticação integrada e escolha o banco padrão como sendo o que acabamos de criar.

Vamos criar nesse datamodule dois métodos bastante parecidos com o btVerPropriedades da nossa form principal, um para criar a tabela no banco de dados e o outro para persistir os valores. Coloque a unit TypInfo no uses de seu datamodule.

O método para criar a tabela criará no banco de dados uma tabela com o mesmo nome da classe, através do comando "create table", concatenando as strings para gerar o comando à medida em que vai varrendo as propriedades do objeto. O segundo método varrerá as propriedades do objeto gravando seus valores através de um comando insert.

Como os nomes dos tipos de dados nos diversos bancos e até no próprio Delphi são diferentes, você terá de criar um método para mapear isto. Por exemplo, transformar o tipo string em varchar(255). Criaremos um método que, recebendo um nome de um tipo de dado do Delphi, devolve o nome de um tipo de dado do banco de dados que vamos usar. Com isso poderemos criar tabelas com diferentes tipos de campos através de um objeto que pode ter qualquer tipo de atributo. Esse código pode ser visto na **Listagem 9.**

Listagem 9. Mapeamento de tipos do Delphi para tipos do SQL (uma forma simples)

```
function TdmPersistencia.MapeiaTipo(tipoDelphi: string): string;
begin
  if lowercase(tipoDelphi) = 'string' then
    Result := 'varchar(255)'
  else    if lowercase(tipoDelphi) = 'tdatetime' then
    Result := 'datetime'
  else
    Result := tipoDelphi;
end;
```

Além disso deve tornar o método flexível o suficiente para dar o tamanho certo aos campos, dependendo do nome da propriedade. Assim como é feito no Hibernate e NHibernate, você pode incrementar, criando um arquivo XML ou .ini de mapeamento de Propriedade x Tipo e propriedade x Campo. No exemplo que estamos usando aqui estamos usando tipos fixos, inflexíveis. E cada propriedade será armazenada no campo de mesmo nome na tabela.

Para não duplicarmos código, o que é sempre ruim, vamos mover o método FormataPropriedade da **Listagem 7** para o datamodule, mas vamos alterá-lo para que coloque aspas onde forem necessárias, com o objetivo de montarmos um comando insert. O método alterado deverá parecer com o da **Listagem 10.**

Listagem 10. Function FormataPropriedade alterada

```
// coloque Variants no uses
function TdmPersistencia.FormataPropriedade(Instancia: TObject;
  const PropInfo: TPropInfo): string;
begin
  if (PropInfo.PropType^.Kind = tkFloat) and (PropInfo.
   PropType^.Name = 'TDate') then
    Result := QuotedStr( FormatDateTime('yyyy-mm-dd', GetPropValue
     (Instancia, PropInfo.Name)))
  else  if (PropInfo.PropType^.Kind = tkFloat) and
   (PropInfo.PropType^.Name = 'TTime') then
    Result := QuotedStr( FormatDateTime
     ('hh:nn:ss', GetPropValue(Instancia, PropInfo.Name)))
  else if (PropInfo.PropType^.Kind = tkFloat) and
   (PropInfo.PropType^.Name = 'TDateTime') then
    Result := QuotedStr( FormatDateTime
     ('yyyy-mm-dd hh:nn:ss', GetPropValue(Instancia, PropInfo.Name)))
  else if (PropInfo.PropType^.Kind in [tkString, tkLString,
  tkWString, tkUString {tkUString apenas para Delphi 2009}]) and
  (trim(lowercase(PropInfo.PropType^.Name)) = 'string')  then
    Result := QuotedStr( VarToStr( GetPropValue
     (Instancia, PropInfo.Name)))
  else
    Result := VarToStr( GetPropValue(Instancia, PropInfo.Name));
end;
```

Nota

Nota: A partir do Delphi 2009 existe mais um kind de string chamado tkUString, então este trecho de código da Listagem 10 deve ser mudado para:

```
if (PropInfo.PropType^.Kind in [tkString, tkLString, tkWString, tkUString]) and
(trim(lowercase(PropInfo.PropType^.Name)) = 'string') then
```

Caso isso não seja feito o SQL resultante não envolverá as strings com aspas simples, como em "Clube Delphi".

Nota

Não estamos tratando o caso de campos float, double, extended ou currency simplesmente porque nossas classes até agora não têm nenhum campo deste tipo.

Devemos agora fazer dois métodos: um para criar uma tabela no banco de dados a partir de um objeto e outro para gravar no banco o estado desse objeto, persistindo-o. Como a tabela terá o mesmo número de colunas que o objeto tem de propriedades publicadas, o valor de cada propriedade ficará gravado em cada uma das colunas.

Nossa tabela também terá um ID que será a chave primária. O método CriaTabela deve varrer todas as informações sobre propriedades do objeto e concatenar uma string com o comando Create Table. Quando a string com os metadados da tabela estiver completa, basta executá-la que a tabela será criada no banco de dados.

Já o método SalvaObjeto fará quase a mesma coisa que o CriaTabela, mas não se preocupando muito com os tipos dos dados do objeto e sim com os valores dos dados em si. Com base nisso o método concatenará uma string com um comando insert into que quando completo inserirá um registro na tabela.

Duas coisas importantes não estamos fazendo aqui: se uma das propriedades de um objeto é um outro objeto, deveríamos varrer o objeto conectado a ele, recursivamente e criar outra tabela, talvez com uma chave estrangeira e algum relacionamento e também popular um registro com as propriedades desse objeto conectado.

Além disso estamos dando insert direto na tabela sem se preocupar com o controle de transações. Para casos em que um pedido deve ser salvo com todos os itens, devemos implementar esse controle de transações. Como nosso objetivo aqui não é criar um framework de persistência completo, mas apenas um modelo didático e uma forma de o usuário poder configurar mais detalhes de sua interface, não nos preocuparemos com essas questões no momento.

Implemente o método CriaTabela conforme a **Listagem 11** e o método SalvaObjeto conforme a **Listagem 12** (lembre-se que eles devem ser declarados na interface como públicos).

Nota

Como estamos usando a função VarToStr, não esqueça de colocar variants no uses do dmPersistencia.

Listagem 11. Método CriaTabela

```
procedure TdmPersistencia.CriaTabela(obj: TPersistent);
var
  infoProps: TPropInfo;
  plistaProps: PPropList;
  listaProps: TPropList;
  i: integer;
  totalProps: integer;
  comando: string;
  listaTabelas: TStringList;
  idx: integer; //índice da tabela na lista de tabelas do banco
begin

  //sempre inicialize o ponteiro como nil
  plistaProps := nil;

  listaTabelas := TStringList.Create;
  listaTabelas.Sorted := true;
  ADOConn.GetTableNames(listaTabelas);

  try
    //cria a tabela apenas se ela não existir.
    if not listaTabelas.Find(obj.ClassName, idx) then
    begin
      comando := 'create table '+obj.ClassName+'( '+#13#10;
      totalProps := GetPropList(obj, plistaProps);
      listaProps := plistaProps^;
      for I := 0 to totalProps - 1 do
      begin
        infoProps := listaProps[i]^;
        comando := comando + infoProps.Name+ ' ' +
          MapeiaTipo(infoProps.PropType^.Name)+', '+ #13#10
      end;
      comando := comando + 'constraint pk_'+obj.ClassName+
        ' primary key (ID)' + #13#10 + ')';

      ADOCom.CommandText := comando;
      ADOCom.Execute;
    end;

    {
     caso a tabela já exista, colocar aqui uma verificação campo a campo.
     qualquer campo que exista no objeto e não na tabela deve ser transformado
      numa nova instrução.
    }

  finally
    ADOConn.Close;
    listaTabelas.Free;
    //não podemos esquecer de liberar a memoria do ponteiro
    FreeMem(plistaProps);
  end;

end;
```

Listagem 12. Método SalvaObjeto

```
procedure TdmPersistencia.SalvaObjeto(obj: TPersistent);
var
  infoProps: TPropInfo;
  plistaProps: PPropList;
  listaProps: TPropList;
  i: integer;
  totalProps: integer;
  campos: string;
  valores: string;
  comando: string;
begin

  //sempre inicialize o ponteiro como nil
  plistaProps := nil;

  totalProps := GetPropList(obj, plistaProps);
  listaProps := plistaProps^;

  try

    for I := 0 to totalProps - 1 do
    begin
      infoProps := listaProps[i]^;
      if i = totalProps-1 then
      begin
        campos := campos + infoProps.Name+ ' ';
       valores := valores + FormataPropriedade(obj, infoProps) + ' ';
      end
      else
      begin
        campos := campos + infoProps.Name+ ', ';
       valores := valores + FormataPropriedade(obj, infoProps) + ', ';
      end;
    end;

    comando := 'insert into '+obj.ClassName+'( '+
      campos + ') values ('+ valores +');';

    ADOCom.CommandText := comando;
    ADOCom.Execute;
  finally
    ADOConn.Close;
    //não podemos esquecer de liberar a memoria do ponteiro
    FreeMem(plistaProps);
  end;

end;
```

O método CriaTabela irá varrer um objeto criando os metadados no banco baseado nas propriedades do objeto, criando uma tabela com o mesmo nome de sua classe caso não existir, enquanto que o método SalvaObjeto criará um registro nesta tabela salvando o valor de cada propriedade na sua respectiva coluna.

Vale ressaltar aqui que todas as nossas classes neste framework obrigatoriamente deverão ter uma propriedade chamada ID, do tipo integer, e que esta sempre será a chave primária de nossas tabelas. Estamos criando um contexto onde a chave é criada pelo sistema ou pelo desenvolvedor e atribuída à propriedade ID do objeto, porém o mais correto seria essa propriedade ser somente leitura e o ID ser um campo Auto Incremento gerado no banco. O objeto deveria ter esse campo setado apenas depois de salvo.

Volte para a frmTesteRTTI e adicione mais dois botões, um chamado btCriaTabela e outro Chamado btCriaRegistro conforme a **Figura 3.** No evento OnClick do botão btCriaTabela nós iremos criar um objeto do tipo TCliente e executar o método CriaTabela do datamodule, conforme a **Listagem 13.**

Listagem 13. Criando a tabela

```
// uses uDmPersistencia
procedure TfrmTesteRTTI.btCriaTabelaClick(Sender: TObject);
var
  obj: TCliente;
begin
  obj := TCliente.Create;
  try
    dmPersistencia.CriaTabela(obj);
    ShowMessage('Tabela Criada!');
  finally
    obj.Free;
  end;
end;
```

Isso deverá criar a tabela de clientes no banco de dados. No botão "Criar Registro" iremos criar alguns objetos TCliente, setar suas propriedades e gravá-los no banco de dados. Digite no seu evento OnClick o código da **Listagem 14.**

O que fizemos foi criar cinco objetos TCliente's diferentes, atribuir valores a seus atributos e usar o método SalvaObjeto do dmPersistencia para salvar cada um desses objetos na tabela. Esse SalvaObjeto é o método da **Listagem 12**, que cria na tabela um registro com o estado do objeto.

Execute o programa, clique primeiro no botão "Criar Tabela" e depois no "Criar Registro", nessa ordem. A tabela de clientes

será criada, e serão adicionados cinco registros, como era de se esperar. Você vê o resultado do MS SQL Server Management Studio na **Figura 4.**

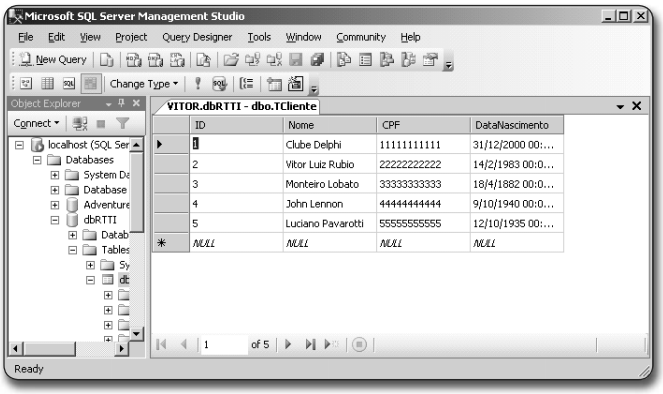


Figura 4. Nossos cinco registros adicionados

Listagem 14. Criando os registros

```
procedure TfrmTesteRTTI.btCriaRegistroClick(Sender: TObject);
var
  obj1,obj2,obj3,obj4,obj5: TCliente;
begin

  obj1 := TCliente.Create;
  obj1.ID := 1;
  obj1.Nome := 'Clube Delphi';
  obj1.CPF := '11111111111';
  obj1.DataNascimento := StrToDate('31/12/2000');

  obj2 := TCliente.Create;
  obj2.ID := 2;
  obj2.Nome := 'Vitor Luiz Rubio';
  obj2.CPF := '22222222222';
  obj2.DataNascimento := StrToDate('14/02/1983');

  obj3 := TCliente.Create;
  obj3.ID := 3;
  obj3.Nome := 'Monteiro Lobato';
  obj3.CPF := '33333333333';
  obj3.DataNascimento := StrToDate('18/04/1882');

  obj4 := TCliente.Create;
  obj4.ID := 4;
  obj4.Nome := 'John Lennon';
  obj4.CPF := '44444444444';
  obj4.DataNascimento := StrToDate('9/10/1940');

  obj5 := TCliente.Create;
  obj5.ID := 5;
  obj5.Nome := 'Luciano Pavarotti';
  obj5.CPF := '55555555555';
  obj5.DataNascimento := StrToDate('12/10/1935');

  try
    dmPersistencia.SalvaObjeto(obj1);
    dmPersistencia.SalvaObjeto(obj2);
    dmPersistencia.SalvaObjeto(obj3);
    dmPersistencia.SalvaObjeto(obj4);
    dmPersistencia.SalvaObjeto(obj5);
  finally
    obj1.Free;
    obj2.Free;
    obj3.Free;
    obj4.Free;
    obj5.Free;
  end;

  ShowMessage('Registros Criados!');

end;
```

Obviamente que esta implementação está longe de ser um framework ou camada de persistência, ou um DAO. Mesmo assim ele ensina as bases de como se fazer um. Ainda precisa ser implementado o update de registros existentes, relacionamentos, conflitos de chaves, exclusão, mapeamentos array x Banco / banco x array, TObjectList x Banco / Banco x TObjectList e mapeamentos similares para ClientDataset, por exemplo.

Vamos criar agora uma forma de carregar um objeto único do banco de dados a partir de seu ID. Para isso deveremos passar o ID para uma query que nos retornará um registro no banco de dados. O método deverá criar um objeto do tipo esperado e fazer o processo inverso do método SalvaObjeto para que nos traga uma instância da classe correta e com todas as propriedades setadas corretamente, cada propriedade com um valor da tabela. Coloque um ADODataset no seu datamodule chamado adoObj. Crie no datamodule uma procedure que receba um objeto TPersistent como parâmetro e um integer. Essa procedure fará um select no banco de dados e populará o objeto com o resultado do banco, que estará no adoObj.

Sete a propriedade connection do adoObj como adoConn. A procedure para consulta segue na **Listagem 15** (lembre-se que ele deve ser declarado na interface como público).

Listagem 15. Recuperando objeto

```
procedure TdmPersistencia.RecuperaObjeto(obj: TPersistent; id:
integer);
var
  comando: string;
  infoProps: TPropInfo;
  plistaProps: PPropList;
  listaProps: TPropList;
  i: integer;
  totalProps: integer;
begin

  //sempre inicialize o ponteiro como nil
  plistaProps := nil;

  comando := 'select * from ' + obj.ClassName + ' where ID = '
   +IntToStr(id);
  ADOObj.CommandText := comando;

  try
    ADOObj.Open;

    //se não tiver registros sai sem fazer nada
    if ADOObj.RecordCount = 0 then
      exit;

    totalProps := GetPropList(obj, plistaProps);
    listaProps := plistaProps^;
    for I := 0 to totalProps - 1 do
    begin
      infoProps := listaProps[i]^;
      SetPropValue(obj, infoProps.Name, ADOObj.FieldByName
       (infoProps.Name).Value);
    end;

  finally
    ADOObj.Close;
    ADOConn.Close;
    //não podemos esquecer de liberar a memória do ponteiro
    FreeMem(plistaProps);
  end;

end;
```

Volte para o form principal e coloque mais um botão chamado btCliente e um Edit chamado edIdCli conforme a **Figura 3.** No evento OnClick do btCliente coloque o código para chamar a procedure criada na **Listagem 15,** passando o ID que estiver no edIdCli.Text como parâmetro, conforme a **Listagem 16.**

Se você clicar nesse botão, será criado um objeto do tipo TCliente novo e ele será passado para o método RecuperaObjeto. Este se encarregará de popular as propriedades da instância obj.

Listagem 16. Testando a recuperação de objeto por ID

```
procedure TfrmTesteRTTI.btClienteClick(Sender: TObject);
var
  obj: Tcliente;
begin

  obj := TCliente.Create;
  try
    dmPersistencia.RecuperaObjeto(obj, strtoint(edIdCli.Text));
    if obj.ID <> 0 then  //para saber se recuperou algo
      ShowMessage(obj.Nome + ' Recuperado com sucesso');
  finally
    obj.Free;
  end;

end;
```

Cópia de objetos

Podemos criar com as técnicas vistas até aqui uma maneira de clonar objetos, mas não é só isso. Usando RTTI podemos "Copiar" quaisquer objetos que implementem uma mesma interface, ou que tenha as mesmas propriedades. Vamos a um exemplo. Criaremos no nosso datamodule uma procedure que aceite dois objetos do tipo TPersistent como parâmetros. Os dois objetos devem entrar já criados. O código deste método nada mais faz do que varrer a lista de propriedades que for menor, copiando os valores do objeto 1 para o objeto 2.

Criaremos o método CopiaObjetos (público) que copiará um objeto A para um objeto B propriedade por propriedade. Os objetos não precisam ser da mesma classe ou de classes parentes, basta que as propriedades tenham o mesmo nome. A **Listagem 17** mostra como isso é feito.

Listagem 17. Copiando objetos

```
procedure TdmPersistencia.CopiaObjetos(obj1, obj2: TPersistent);
var
  infoProps: TPropInfo;
  plistaProps1: PPropList;
  plistaProps2: PPropList;
  listaProps: TPropList;
  i: integer;
  totalProps1: integer;
  totalProps2: integer;
  totalPropsMenor: integer;
begin

  plistaProps1 := nil;
  plistaProps2 := nil;

  try
    totalProps1 := GetPropList(obj1, plistaProps1);
    totalProps2 := GetPropList(obj2, plistaProps2);

    if totalProps1 < totalProps2 then
    begin
      totalPropsMenor := totalProps1;
      listaProps := plistaProps1^;
    end
    else
    begin
      totalPropsMenor := totalProps2;
      listaProps := plistaProps2^;
    end;

    for I := 0 to totalPropsMenor - 1 do
    begin

      infoProps := listaProps[i]^;
      //se for uma propriedade comum aos 2
      if (IsPublishedProp(obj1, infoProps.Name))and
        (IsPublishedProp(obj2, infoProps.Name)) then
         SetPropValue(obj2, infoProps.Name, GetPropValue
         (obj1, infoProps.Name));
    end;


  finally
    FreeMem(plistaProps1);
    FreeMem(plistaProps2);
  end;

end;
```

O funcionamento é simples: temos dois objetos, então extraímos duas listas de propriedades. Então varremos a lista de propriedades menor e para cada nome de propriedade do objeto A, verificamos se existe propriedade com o mesmo nome no objeto B. Com isso podemos até copiar a propriedade caption de um label para um form, por exemplo. Falta aqui colocar uma lista de propriedades ignoradas, deixo isso a cargo do leitor. Talvez aconteça de duas propriedades de mesmo nome serem de tipos diferentes, mas isso é raro. Nenhum programador de componentes daria propositalmente o nome de "Text" a um atributo do tipo integer. Mas pode acontecer, e como o universo do Delphi é muito grande, com muitas bibliotecas de componentes, não custa nada fazer essa verificação.

Para criar uma cópia idêntica de um objeto basta executar esse método, passando como parâmetros os dois objetos já criados, a origem é o primeiro e o destino é o segundo. Crie mais um botão no form principal e codifique o seu evento OnClick conforme a **Listagem 18.**

Listagem 18. Usando o método de cópia

```
procedure TfrmTesteRTTI.btClonarClick(Sender: TObject);
var
  obj1, obj2: TCliente;
begin
  obj1 := TCliente.Create;
  obj2 := TCliente.Create;

  obj1.ID := 6;
  obj1.Nome := 'Fulano de tal';
  obj1.CPF := '666666666666';
  obj1.DataNascimento := StrToDate('31/12/2000');

  try
    dmPersistencia.CopiaObjetos(obj1, obj2);

    ShowMessage(
      'Pode até parecer o mesmo objeto: ' +#13#10 +
      'ID: ' + inttostr(obj1.ID) + #13#10 +
      'Nome: ' + obj1.Nome + #13#10 +
      'CPF: ' + obj1.CPF + #13#10 +
      'Nascimento: ' + DateTimeToStr( obj1.DataNascimento)
    );

    ShowMessage(
      'Pode até parecer o mesmo objeto: ' +#13#10 +
      'ID: ' + inttostr(obj2.ID) + #13#10 +
      'Nome: ' + obj2.Nome + #13#10 +
      'CPF: ' + obj2.CPF + #13#10 +
      'Nascimento: ' + DateTimeToStr( obj2.DataNascimento)
    );

    obj1.ID := 10;
    obj1.Nome := 'Beltrano de tal';
    obj1.CPF := '999999999999';
    obj1.DataNascimento := StrToDate('31/12/2009');

    ShowMessage(
      'Mas trocando os valores você percebe que não é: ' +#13#10 +
      'ID: ' + inttostr(obj1.ID) + #13#10 +
      'Nome: ' + obj1.Nome + #13#10 +
      'CPF: ' + obj1.CPF + #13#10 +
      'Nascimento: ' + DateTimeToStr( obj1.DataNascimento)
    );

    ShowMessage(
      'Mas trocando os valores você percebe que não é: ' +#13#10 +
      'ID: ' + inttostr(obj2.ID) + #13#10 +
      'Nome: ' + obj2.Nome + #13#10 +
      'CPF: ' + obj2.CPF + #13#10 +
      'Nascimento: ' + DateTimeToStr( obj2.DataNascimento)
    );
  finally
    obj1.Free;
    obj2.Free;
  end;

end;
```

O código da **Listagem 18** faz um teste passo a passo da cópia de dois objetos, mostrando mensagens com os atributos do mesmo, provando que são objetos distintos em vez de duas referências ao mesmo objeto. Mudando-se um deles o outro permanece, como um legítimo clone.

Para exemplificar como um objeto pode ser apresentado em sua GUI, vamos criar uma form para apresentação de Clientes. Crie uma nova form com o nome frmClienteGui e salve sua unit como uFrmClienteGui. Coloque três Edit's, um DateTimePicker e um botão conforme a **Figura 5.**

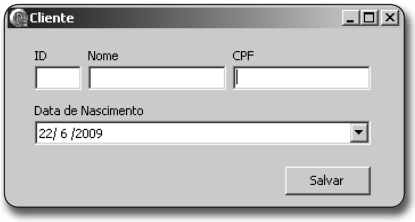


Figura 5. GUI de Clientes

Dê os nomes de edId, edNome, edCpf, dtpData e btSalvar aos componentes, respectivamente. Para diminuir ao máximo o acoplamento desta form com a unit uCliente temos três opções: criar o objeto a ser salvo através de factory method (um famoso padrão de projeto), passar um cliente já criado para a form através de parâmetros ou propriedades, mantendo-o como uma variável do tipo TPersistent, ou também podemos salvar fora da form. Neste exemplo é isso que faremos.

Crie neste mesmo form um campo boolean na seção private chamado FSalvar. Crie uma propriedade boolean chamada salvar que leia seu valor de FSalvar. O código do botão "Salvar" simplesmente muda o valor de FSalvar para true.

Implemente na form propriedades com os mesmos nomes que as propriedades de TCliente, onde seus valores são armazenados ou exibidos nos componentes correspondentes conforme o código da **Listagem 19.**

Quem apenas programa "da maneira RAD de ser" do Delphi talvez não esteja familiarizado com todos esses conceitos de classes, properties, métodos e assim por diante, mas quem já conhece um pouco de POO sabe exatamente o que estamos fazendo: criando uma interface com o usuário que é apenas a interface gráfica, mais nada, sem código ou lógica de negócio espalhados pelos eventos dos componentes, a não ser que seja lógica de visualização e apresentação específica deste form.

Podemos ir mais além: essa GUI tem propriedades com os mesmos nomes das propriedades da classe de clientes, então podemos copiar um objeto cliente para uma instância "vazia" do form e o objeto será exibido, e podemos fazer o inverso,

Listagem 19. Implementação do frmClienteGui

```
unit uFrmClienteGui;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ComCtrls;

type
  TfrmClienteGui = class(TForm)
    edId: TEdit;
    edNome: TEdit;
    edCpf: TEdit;
    dtpData: TDateTimePicker;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    btSalvar: TButton;
    procedure btSalvarClick(Sender: TObject);
  private
    FSalvar: boolean;
  protected
    //gets
    function GetID: integer;  virtual;
    function GetNome: string; virtual;
    function GetCPF: string;  virtual;
    function GetDataNascimento: TDateTime; virtual;
    //sets
    procedure SetID(const valor: integer);   virtual;
    procedure SetNome(const valor: string); virtual;
    procedure SetCPF(const valor: string);  virtual;
    procedure SetDataNascimento(const valor: TDateTime);  virtual;
  public
    property Salvar: boolean read FSalvar;
  published
    property ID: integer read GetID write SetID;
    property Nome: string read GetNome write SetNome;
    property CPF: string read GetCPF write SetCPF;
    property DataNascimento: TDateTime read GetDataNascimento
    write SetDataNascimento;
  end;

var
  frmClienteGui: TfrmClienteGui;

implementation

{$R *.dfm}

procedure TfrmClienteGui.btSalvarClick(Sender: TObject);
begin
  FSalvar := true;
  close;
end;

function TfrmClienteGui.GetCPF: string;
begin
  Result := edCpf.Text;
end;

function TfrmClienteGui.GetDataNascimento: TDateTime;
begin
  Result := dtpData.DateTime;
end;

function TfrmClienteGui.GetID: integer;
begin
  Result := StrToInt(edId.Text);
end;

function TfrmClienteGui.GetNome: string;
begin
  Result := edNome.Text;
end;

procedure TfrmClienteGui.SetCPF(const valor: string);
begin
  edCpf.Text := valor;
end;

procedure TfrmClienteGui.SetDataNascimento(const valor:
TDateTime);
begin
  dtpData.DateTime := valor;
end;

procedure TfrmClienteGui.SetID(const valor: integer);
begin
  edId.Text := IntToStr(valor);
end;

procedure TfrmClienteGui.SetNome(const valor: string);
begin
  edNome.Text := valor;
end;

end.
```

copiando o form para um objeto cliente e o mesmo terá as propriedades preenchidas todas de uma vez.

Agora a parte mais interessante: no form principal clique no menu file > use unit e use a unit da frmClienteGui. Vamos criar na form principal mais um botão, que será responsável por criar um cliente, exibi-lo na sua GUI, e salvá-lo no banco de dados caso o usuário tenha clicado em salvar. Iremos usar o nosso método de cópia duas vezes, uma para copiar o objeto para a GUI e a outra para copiar a GUI para o objeto. Tome por base o layout da **Figura 3** para finalizar a construção do form principal. O botão "Cria e Apresenta" foi codificado conforme a **Listagem 20.**

Listagem 20. Código do botão "Cria e Apresenta"

```
procedure TfrmTesteRTTI.btApresentarClick(Sender: TObject);
var
  obj1: TCliente;
  frm: TfrmClienteGui;
begin
  obj1 := TCliente.Create;
  obj1.ID := 20;
  obj1.Nome := 'Albert Einstein';
  obj1.CPF := '77777777777';
  obj1.DataNascimento := StrToDate('14/03/1879');

  frm := TfrmClienteGui.Create(nil);

  try
    dmPersistencia.CopiaObjetos(obj1, frm); //carrega a form com
     informações do objeto
    frm.ShowModal;
    if frm.Salvar then
    begin
      dmPersistencia.CopiaObjetos(frm, obj1); //faz o caminho
       inverso, já que o form contém informações mais atualizadas
      dmPersistencia.SalvaObjeto(obj1);
      ShowMessage('Objeto Salvo!');
    end;
  finally
    frm.Free;
    obj1.Free;
  end;

end;
```

Veja como podemos usar três funcionalidades do nosso sistema para uma mesma entidade com acoplamento zero. Podemos criar um cliente, apresentá-lo, modificá-lo e salvá-lo sem que a GUI conheça as classes de negócio ou o DataModule, e vice-versa.

Aplicando padrões de projeto nessas técnicas podemos criar um ERP multibanco, super flexível e super rápido de se alterar, e o mais importante: puramente orientado a objetos, mas sem perder as facilidades do RAD.

Salvando propriedades de componentes

Para esta etapa vamos criar dois novos forms, um será a nossa "cobaia" e a outra será o nosso "Object Inspector" (depois de criar cada um destes formulários, retire-os da sessão de criação automática nas opções de projeto, pois serão criados em runtime). Nossa cobaia será um exemplo de form onde o usuário final poderá mover e alterar algumas propriedades dos componentes. O nosso "Object Inspector" será bem limitado, possuindo a princípio apenas sete propriedades, sendo elas: Top, Left, Color, Width, Height, Text e Caption. Qualquer componente que possuir uma dessas propriedades poderá ser editado e salvo.

Há várias maneiras de se fazer isso, e você pode implementar a sua. Faremos da maneira mais flexível/extensível possível, normalizada em vez de um único "tabelão", com a desvantagem de ter o maior tempo de implementação. Talvez não seja a maneira mais performática ou correta, mas serve para fins didáticos.

Aqui cabe uma observação: há uma certa dificuldade em se persistir propriedades que são referências a outros objetos. Justamente por isso que não estamos persistindo a fonte. Para se fazer isso seria necessário fazer uma outra chamada, recursiva, ao método de persistência passando a fonte como objeto. Isso criaria uma complexa relação de componente x subcomponente.

Além disso, há também certas dificuldades em se persistir propriedades que sejam records, arrays, sets, enumerações entre outros. Embora difícil, não é impossível. O próprio ambiente do Delphi faz isso, usando as units de design time designIntf e designEditors além da TypInfo. Isso é muito útil na criação de componentes ou plug-ins para o próprio Delphi, quando precisarmos fazer editores de propriedade e de componente específicos.

Primeiro de tudo vamos criar as nossas tabelas. Teremos quatro: Formularios, Componentes, Propriedades e Valores. A tabela Valores é uma tabela relacional que conterá o cruzamento Tabela x Componente x Propriedade e o valor da propriedade. A **Listagem 21** contém todas as tabelas que precisaremos, e insere na tabela propriedades apenas os nomes das propriedades que permitiremos que sejam alteradas, caso existirem.

Listagem 21. Tabelas do banco de dados

```
create table [Formularios]
(
CodForm integer not null primary key identity(1,1),
NomeForm varchar(255),
ClasseForm varchar(255)
)

create table [Componentes]
(
CodComp integer not null primary key identity(1,1),
CodForm integer foreign key references Formularios(CodForm),
NomeComp varchar(255),
ClasseComp varchar(255)
)

create table [Propriedades]
(
CodProp integer not null primary key identity(1,1),
NomeProp varchar(255),
TipoProp varchar(255)
)

create table [Valores]
(
CodValor integer not null primary key identity(1,1),
CodForm integer foreign key references Formularios(codForm),
CodComp integer foreign key references Componentes(codComp),
CodProp integer foreign key references Propriedades(codProp),
Valor varchar(255)
)

insert into Propriedades values ('Top', 'integer');
insert into Propriedades values ('Left', 'integer');
insert into Propriedades values ('Color', 'TColor');
insert into Propriedades values ('Caption', 'string');
insert into Propriedades values ('Text', 'string');
insert into Propriedades values ('Width', 'integer');
insert into Propriedades values ('Height', 'integer');
```

Vamos criar um novo formulário chamado frmObjectInspector e salvar sua unit como uFrmObjectInspector.pas. Nesta form vamos colocar cinco ADODataset's, cinco DatasetProvider's e cinco ClientDataset's.

Ligue os cinco ADODataset's ao dmPersistencia.adoConn. O nome de cada adoDataset corresponderá a cada uma das tabelas que nós criamos, exceto o último que terá o nome de adoInspector. Então teremos um adoFormularios, adoComponentes, adoPropriedades, adoValores e adoInspector. Os cinco DatasetProviders seguirão a mesma linha: dspFormularios, dspPropriedades, dspComponentes, dspValores e dspInspector.

Cada um terá em sua propriedade DataSet o AdoDataset correspondente. Os cinco ClientDataset's serão cdsFormularios, cdsPropriedades, cdsComponentes, cdsValores e cdsInspector. Eles terão em suas propriedades ProviderName o nome do Provider correspondente. Para facilitar, coloquei na **Listagem 22** o código DFM que representa o relacionamento entre os componentes.

Listagem 22. Relacionamento entre os componentes

```
object frmObjectInspector: TfrmObjectInspector
  …
  object adoFormularios: TADODataSet
    Connection = DmPersistencia.adoConn
  end
  object adoComponentes: TADODataSet
    Connection = DmPersistencia.adoConn
  end
  object adoPropriedades: TADODataSet
    Connection = DmPersistencia.adoConn
  end
  object adoValores: TADODataSet
    Connection = DmPersistencia.adoConn
  end
  object adoInspector: TADODataSet
    Connection = DmPersistencia.adoConn
  end
  object dspFormularios: TDataSetProvider
    DataSet = adoFormularios
  end
  object dspPropriedades: TDataSetProvider
    DataSet = adoPropriedades
  end
  object dspComponentes: TDataSetProvider
    DataSet = adoPropriedades
  end
  object dspValores: TDataSetProvider
    DataSet = adoValores
  end
  object dspInspector: TDataSetProvider
    DataSet = adoInspector
  end
  object cdsFormularios: TClientDataSet
    ProviderName = 'dspFormularios'
  end
  object cdsPropriedades: TClientDataSet
    ProviderName = 'dspPropriedades'
  end
  object cdsComponentes: TClientDataSet
    ProviderName = 'dspComponentes'
  end
  object cdsValores: TClientDataSet
    ProviderName = 'dspValores'
  end
  object cdsInspector: TClientDataSet
    ProviderName = 'dspInspector'
  end
end
```

Enquanto cada ClientDataset servirá para ler ou escrever dados da tabela correspondente, o cdsInspector servirá para consultar e editar as propriedades, trazendo todas de um determinado componente e form. Este último terá dois campos do tipo LookUp Chamados "Propriedade" e "Tipo" ligados ao cdsPropriedades, que servirão para exibir o nome da propriedade em um DbGrid, ao invés de exibir códigos, e para podermos saber de que tipo que é uma propriedade. As propriedades CommandText de cada um dos 5 adoDataset's podem ser vistas na **Listagem 23**.

Listagem 23. CommandText de cada um dos AdoDataset's

```
adoFormularios:
select * from Formularios where nomeForm = :nomeForm and
classeForm = :classeForm

adoComponentes:
select * from Componentes where codForm = :codForm and
nomeComp = :nomeComp and classeComp = :classeComp

adoPropridades:
select * from Propriedades

adoValores:
select CodValor, codForm,  codComp,  codProp, valor
from Valores
where  codForm = :codForm and  codComp = :codComp and
codProp = :codProp

adoInspector:
select * from Valores
where codForm = :codForm and codComp = :codComp
```

Nota

Para adicionar os Fields, na propriedade Parameters de cada ADODataSet, configure o DataType de cada parâmetro, sendo string para os campos "nomes" e integer para os campos "códigos".

Adicione os campos TFields nos cinco ClientDataset's. No cdsInspector os campos Propriedade e Tipo são campos lookup que devem ter como LookUpDataSet o cdsPropriedades, KeyField e lookUpKeyField o campo CodProp e como LookupResultFiled o campo correspondente, TipoProp ou NomeProp.

Adicione um DBGrid (dbgInspector) no Object Inspector e um DataSource chamado dsInspector. Aponte a propriedade DataSet do dsInspector para cdsInspector e a propriedade DataSource do dbgInspector para dsInspector. Adicione na dbgInspector apenas os campos Propriedade e Valor, sendo o Tipo opcional. Coloque um botão chamado btSalvar com o caption igual a "Salvar". O form ficará com o aspecto da **Figura 6.**

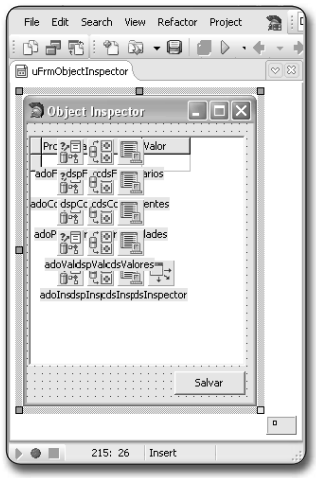


Figura 6. "Object Inspector"

Nota

Neste ponto o leitor notará que poderá usar a abordagem anterior, de criar a tabela e popular os registros, para criar esse Object Inspector, bastando para isso criar classes para manipular formulários, componentes, propriedades e a relação de valores. Notará também que poderíamos criar uma tabela de classes para simular um comportamento/visual padrão para todos os componentes da mesma classe e uma tabela de usuários para definir visibilidade e direito de escrita de cada componente, de cada formulário.

Esse Object Inspector será criado por uma form qualquer, recebendo um componente e a própria form como argumentos. Seu fluxo de funcionamento será o contrário de um Object Inspector de verdade: em vez de varrer as propriedades e valores do componente e gravar no banco de dados, nós vamos varrer a tabela propriedades e verificar sua existência no componente. Caso exista nós permitiremos a edição da propriedade e salvaremos os atributos do componente configurado.

Essa abordagem "ao contrário" permite limitarmos as propriedades que o cliente pode alterar, além de ser mais fácil de implementar e mais performático, pois não gravará no banco de dados propriedades que o cliente nunca irá alterar.

No caminho inverso, ao abrir um form qualquer, instanciamos o Object Inspector e varremos a tabela de componentes. Para cada componente nós varremos a tabela de Valores e setamos as respectivas propriedades. É por isso que usamos dois ClientDataset's para a tabela de valores, no cdsValores nós cadastramos as propriedades uma por uma. Seus parâmetros permitem trazer apenas uma propriedade de cada vez, para um componente e form. O cdsInspector possui parâmetros que filtram apenas por form e componente, permitindo que você visualize todas as propriedades cadastradas para esse componente através dos campos lookups. Dessa forma você pode varrer o cdsInspector para obter as informações que usará com o método IsPublishedProp da unit TypInfo.

O nosso fluxo funcionará da seguinte maneira: logo que abrirmos a nossa form "Cobaia", que pode ser qualquer form customizável da nossa aplicação, um método irá varrer todos os componentes da form. Se o formulário não estiver cadastrado, ele será cadastrado. A cada volta do loop que varre os componentes, o componente será cadastrado caso ainda não seja. Depois disso será varrida a tabela de propriedades, e para cada propriedade que o componente da vez tiver, ela será cadastrada na tabela de valores com as chaves estrangeiras de formulários, componentes, propriedades e o valor da propriedade corrente. Depois de todos os componentes novos salvos, será lida toda a configuração do usuário no banco de dados para se redefinir todos os componentes. Ao se abrir a form não será feito edição ou update dos valores armazenados para não sobrescrever as configurações do usuário com os valores default da form, ou com as configurações do programador.

Faça as devidas configurações no DBGrid para que o campo Propriedade seja somente leitura, para ninguém alterá-lo. Ninguém pode excluir ou inserir valores aqui, apenas alterar. O usuário poderá alterar as propriedades de um componente usando essa ferramenta, e ao clicar em "Salvar", os novos valores são salvos no banco de dados, e depois o componente é carregado com os valores novos. Todos os métodos do Object Inspector serão "Alimentados" com um formulário e um componente. Para evitar acoplamentos e flexibilizar o uso você pode armazenar o form e o componente em edição em variáveis privadas dentro do inspector. Através das propriedades Top e Left o usuário pode mover componentes afastando-os do topo e da margem esquerda, respectivamente. Usando as propriedades Width e Height o usuário pode alterar a largura e altura.

Este inspector está longe de ser perfeito. Para implementar as funcionalidades de arrastar componentes e redimensioná-los com o mouse várias funções da API do Windows terão de ser usadas. Você pode aprender a fazer isso no artigo RTTI na prática, do Autor Gustavo Chaurais, na Edição 97 da Clube Delphi (veja Links).

Vamos ao código para entender melhor como tudo isso vai funcionar. Vamos criar algumas funções no nosso object inspector e depois implementá-las. A procedure CadastraForm receberá um formulário como parâmetro e irá cadastrá-lo na tabela de formulários. Para cada componente do formulário será chamado o método CadastraComponente, em um laço, que receberá como parâmetros o formulário e um componente, e este componente será cadastrado no banco de dados, na tabela componentes. Para cada componente as propriedades serão varridas num laço e toda a propriedade que existir na tabela de propriedades terá seu valor cadastrado na tabela de valores.

A procedure EditaComponente permitirá que o usuário altere os valores das propriedades cadastradas na tabela de valores para o formulário e componente correntes, ou seja, os que forem passados como parâmetro. A procedure CarregaComponente irá chamar a ResgataValores, que irá varrer os componentes do formulário fazendo o processo inverso, setando cada propriedade dos componentes com os valores retirados do banco de dados. Crie no inspector os métodos privados e públicos conforme a interface na **Listagem 24**.

Listagem 24. Métodos do Inspector

```
private
  procedure CadastrarForm(aForm: Tform);
  procedure CadastrarComponente(aForm: TForm;
   aComponent: TComponent);
  procedure CadastrarValores(aForm: TForm;
   aComponent: TComponent);
  function ConsultarForm(aForm: Tform):boolean;
  function ConsultarComponente(aForm: TForm;
   aComponent: TComponent):boolean;
  function ConsultarValores(aForm: TForm;
   aComponent: TComponent):boolean;
  procedure ResgatarValores(aForm: TForm;
   aComponent: TComponent);
public
  procedure EditaComponente(aForm: TForm;
   aComponent: TComponent);
  procedure CarregaComponente(aForm: TForm;
   aComponent: TComponent);
  procedure InicializaForm(aForm: TForm);
end;
```

O método CadastraForm faz exatamente o que o nome diz, executando internamente o ConsultaForm para verificar se ele já não está cadastrado. O método CadastrarComponente primeiro verifica se a form e o componente existem, usando o método ConsultaComponente, e caso o componente não exista ele será cadastrado. Os métodos ConsultarValores e CadastrarValores são análogos, fazendo a mesma coisa com os valores das propriedades. O método EditaComponente abre um componente para edição e o método CarregaComponente varre as propriedades no banco de dados e modifica todas as propriedades que o componente possuir. InicializaForm é a procedure que varrerá todos os componentes da form com um loop. Implemente os métodos conforme a **Listagem 25.**

Listagem 25. Implementação dos métodos do Inspector

```
// uses TypInfo
procedure TfrmObjectInspector.CadastrarComponente(aForm: Tform;
aComponent: TComponent);
begin
  if not ConsultarComponente(aForm, aComponent) then
  begin
    cdsComponentes.Open;
    cdsComponentes.Insert;
    cdsComponentesnomeComp.AsString := aComponent.Name;
    cdsComponentesclasseComp.AsString := aComponent.ClassName;
    cdsComponentescodForm.AsInteger := cdsFormularioscodForm.AsInteger;
    cdsComponentes.Post;
    cdsComponentes.ApplyUpdates(0);
    cdsComponentes.Close;
  end;
end;

procedure TfrmObjectInspector.CadastrarForm(aForm: Tform);
begin
  if not ConsultarForm(aForm) then
  begin
    cdsFormularios.Open;
    cdsFormularios.Insert;
    cdsFormulariosnomeForm.AsString := aForm.Name;
    cdsFormulariosclasseForm.AsString := aForm.ClassName;
    cdsFormularios.Post;
    cdsFormularios.ApplyUpdates(0);
    cdsFormularios.Close;
  end;
end;

procedure TfrmObjectInspector.CadastrarValores
 (aForm: TForm; aComponent: TComponent);
begin

  if not cdsPropriedades.Active then
    cdsPropriedades.Open;

    cdsPropriedades.First;
    while not cdsPropriedades.Eof do
  try

    //cadastra o valor da propriedade apenas se o componente a
    suporta
    if IsPublishedProp(aComponent, cdsPropriedadesnomeProp.
    AsString) then
    begin
      //cadastrando o componente pela primeira vez
      if not ConsultarValores(aForm, aComponent) then
      begin
        cdsValores.Open;
        cdsValores.Insert;
        cdsValoresCodForm.AsInteger := cdsFormularioscodForm.AsInteger;
        cdsValoresCodComp.AsInteger := cdsComponentescodComp.AsInteger;
        cdsValoresCodProp.AsInteger := cdsPropriedadescodProp. AsInteger;

        if GetPropInfo(aComponent, cdsPropriedadesnome
        Prop.AsString).PropType^.Name = 'TColor' then
        cdsValoresValor.Value :=  ColorToString(  GetPropValue
           (aComponent, cdsPropriedadesnomeProp.AsString, true))
        else
          cdsValoresValor.Value :=  GetPropValue
           (aComponent, cdsPropriedadesnomeProp.AsString, true);

        cdsValores.Post;
        cdsValores.ApplyUpdates(0);
        cdsValores.Close;
      end;
    end;

  finally
    cdsPropriedades.Next;
  end;

end;

procedure TfrmObjectInspector.CarregaComponente(aForm: TForm;
 aComponent: TComponent);
begin
 ResgatarValores(aForm, aComponent);
end;

function TfrmObjectInspector.ConsultarComponente(aForm: Tform;
  aComponent: TComponent): boolean;
begin
  if (ConsultarForm(aForm)) then
  begin
    cdsComponentes.Close;
    adoComponentes.Parameters.ParamByName('CodForm').
     Value := cdsFormularioscodForm.Value;
    adoComponentes.Parameters.ParamByName('NomeComp').
     Value := aComponent.Name;
    adoComponentes.Parameters.ParamByName('ClasseComp').
     Value := aComponent.ClassName;
    cdsComponentes.Open;
    Result := (cdsComponentes.RecordCount > 0);
  end
  else
    Result := false;
end;

function TfrmObjectInspector.ConsultarForm(aForm: Tform): boolean;
begin
  cdsFormularios.Close;
  adoFormularios.Parameters.ParamByName('NomeForm').Value
  := aForm.Name;
  adoFormularios.Parameters.ParamByName('ClasseForm').Value
  := aForm.ClassName;
  cdsFormularios.Open;
  Result := (cdsFormularios.RecordCount > 0);
end;

function TfrmObjectInspector.ConsultarValores(aForm:
 Tform;  aComponent: TComponent): boolean;
begin
  if ConsultarComponente(aForm, aComponent) then
  begin
    cdsValores.Close;
    adoValores.Parameters.ParamByName('CodForm').Value:=
    cdsFormularioscodForm.Value;
    adoValores.Parameters.ParamByName('CodComp').Value:=
    cdsComponentescodComp.Value;
    adoValores.Parameters.ParamByName('CodProp').Value :=
    cdsPropriedadescodProp.Value;
    cdsValores.Open;
    Result := (cdsValores.RecordCount > 0);
  end
  else
    Result := false;
end;

procedure TfrmObjectInspector.EditaComponente(aForm: TForm;
  aComponent: TComponent);
begin
  if ConsultarComponente(aForm, aComponent) then
  begin
    Caption := aForm.Name + ' --> ' + aComponent.Name;
    cdsInspector.Close;
    adoInspector.Parameters.ParamByName('CodForm').
     Value := cdsFormularioscodForm.Value;
    adoInspector.Parameters.ParamByName('CodComp').
     Value := cdsComponentescodComp.Value;
    cdsInspector.Open;
  end;
end;

procedure TfrmObjectInspector.InicializaForm(aForm: TForm);
var i: integer;
begin
  CadastrarForm(aForm);
  for I := 0 to aForm.ComponentCount - 1 do
  begin
    CadastrarComponente(aForm, aForm.Components[i]);
    CadastrarValores(aForm, aForm.Components[i]);
    ResgatarValores(aForm, aForm.Components[i]);
  end;
end;

procedure TfrmObjectInspector.ResgatarValores(aForm: TForm;
  aComponent: TComponent);
var
  teste: TColor;
begin
  if not cdsPropriedades.Active then
    cdsPropriedades.Open;
    cdsPropriedades.First;
  while not cdsPropriedades.Eof do
  try

    if IsPublishedProp(aComponent, cdsPropriedadesnomeProp.
     AsString) then
    begin

      if ConsultarValores(aForm, aComponent) then
      try
        if GetPropInfo(aComponent, cdsPropriedadesnomeProp.
        AsString).PropType^.Name = 'TColor' then
        SetPropValue(aComponent, cdsPropriedadesnomeProp.
        AsString, StringToColor(cdsValoresvalor.Value))
        else
          SetPropValue(aComponent, cdsPropriedadesnomeProp.
           AsString, cdsValoresvalor.Value);
      except
        //se você precisar de um log, é aqui que deve colocar, mas
         o loop não deve ser interrompido
      end;

    end;

  finally
    cdsPropriedades.Next;
  end;
end;
```

Repare que os nossos AdoDataset's possuem parâmetros. As funções ConsultarForm, ConsultarComponente e ConsultarValores respectivamente setam (desta vez em runtime) os parâmetros dos AdoDataset's para consultar se uma form existe antes de cadastrá-la e caso exista já trazer seus dados. Também fazem o papel de consultar se um componente e seu form existem antes de cadastrá-los e já trazer os dados dos componentes e consultar se os valores das propriedades de um determinado componente já existem e então trazê-los, para que possam ser editados. Tudo o que vamos cadastrar ou editar tentamos abrir antes, para ver se existe e já trazer os registros corretos.

Merece consideração especial a procedure InicializaForm. Ela executa alguns métodos em uma determinada ordem para assegurar que componentes novos inseridos pelo programador sejam cadastrados no banco de dados antes que sejam disponibilizados para edição pelo usuário, e não permitir que um componente seja cadastrado duas vezes. Além disso, como durante a inicialização do form, os componentes estão com todos os valores padrão, definidos pelo programador, a procedure InicializaForm chama o método ResgatarValores para já trazer os valores anteriormente editados pelo usuário. Isso garante que tanto o programador pode inserir componentes novos no formulário com seus valores default como o usuário pode alterar os novos componentes.

No evento OnClick do btSalvar nós daremos um post e um ApplyUpdates(0) no cdsInspector. Post para que a última linha em edição seja salva e ApplayUpdates(0) para persistir fisicamente as informações na tabela, não tolerando nenhuma falha. Você pode ver na **Listagem 26.**

Listagem 26. Botão Salvar

```
procedure TfrmObjectInspector.btsalvarClick
 (Sender: TObject);
begin
  if (cdsInspector.State in [dsEdit]) then
    cdsInspector.Post;
  cdsInspector.ApplyUpdates(0);
  Close;
end;
```

Com um clique duplo no DBGrid, no campo Valor, mude a propriedade ButtonStyle para cbsEllipsis. Isso fará com que, quando em edição, esse campo tenha aquele botão com reticências. Programe o evento OnEditButtonClick conforme a **Listagem 27.**

Esse evento fará um tratamento especial na hora de mudar a propriedade Color dos componentes, verificando se o tipo da propriedade é um TColor. Caso for, ele irá abrir um TColorDialog para o usuário escolher a cor.

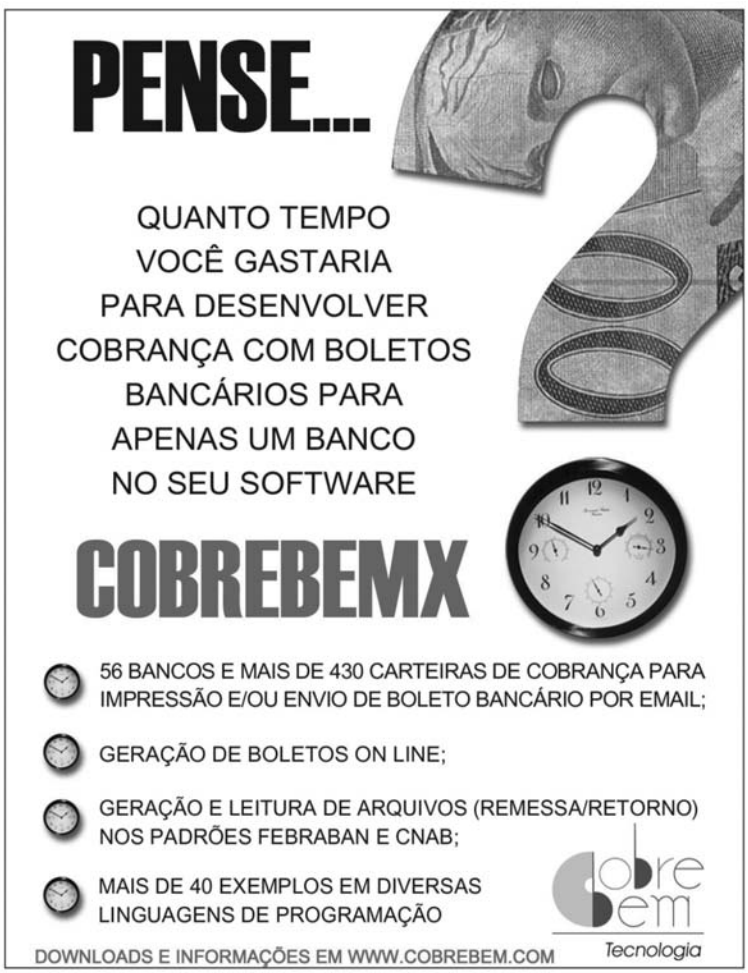
Listagem 27. Editando Cores

```
procedure TfrmObjectInspector.dbgInspectorEditButtonClick(Sender: TObject);
begin
  if not (cdsInspector.State in [dsEdit]) then
    cdsInspector.Edit;
  if cdsInspectorTipo.AsString = 'TColor' then
  begin
    with TColorDialog.Create(nil) do
    try
      if Execute then
        cdsInspectorValor.AsString := ColorToString(Color);
    finally
      Free;
    end;
  end;
end;
```

Salve tudo, porque agora vamos criar o nosso form "cobaia". Abra a frmCobaia criada anteriormente na unit uCobaia e no menu file > use unit coloque a unit uFrmObjectInspector no uses de frmCobaia. Coloque no evento OnActivate o código da **Listagem 28**.

Listagem 28. Abrindo, salvando e carregando configurações do formulário

```
procedure TfrmCobaia.FormActivate(Sender: TObject);
begin
  with TfrmObjectInspector.Create(nil) do
  try
    InicializaForm(self);
  finally
    Release;
  end;
end;
```



No evento OnActivate da frmCobaia nós instanciamos um TFrmObjectInspector e inicializamos o form, para poder salvar os componentes novos e resgatar os valores dos já editados pelo usuário. Fizemos isso no evento OnActivate por dois motivos: primeiro porque no evento OnCreate o formulário ainda não existe e os componentes ainda não têm um Parent, por isso configurações de tamanho serão ignoradas. Segundo porque uma configuração feita pelo administrador ou por outro usuário poderia aparecer automaticamente num sistema em execução ao se minimizar e restaurar a form.

Nota

Para implementarmos a funcionalidade de cada usuário diferente do sistema ter a sua configuração particular nos forms, devemos criar uma tabela de usuários e na tabela Valores colocar a chave estrangeira de Usuários. Não seria interessante você estar usando um formulário e repentinamente os controles mudarem de posição e cor.

Coloque alguns componentes aleatórios da aba standard no seu form cobaia. Eu coloquei dois Edit's, dois Label's e um Button. Crie o método privado AbrirComponente. Este método será responsável por instanciar o Object Inspector, criar os cadastros dos componentes que ainda não estiverem cadastrados e redefinir as propriedades conforme os registros dos usuários, se houver. Este método pode ser visto na **Listagem 29.**

Listagem 29. Método AbrirComponente

```
procedure TfrmCobaia.AbrirComponente(aComponent: TComponent);
begin
  with TfrmObjectInspector.Create(nil) do
  try
    EditaComponente(self, aComponent);
    ShowModal;
    CarregaComponente(self, aComponent);
  finally
    Release;
  end;
end;
```

Executamos o método EditaComponente primeiro, para trazer os valores das propriedades do referido componente e pô-las em modo de edição. Depois executamos ShowModal para que o usuário não abra dois ObjectInspectors ao mesmo tempo, nem mecha nos componentes da form enquanto o inspector estiver ativo. Depois executamos CarregaComponente para configurar o componente conforme o que o usuário modificou.

Selecione todos os componentes de uma vez e defina um evento OnMouseDown que será o mesmo para todos. Você também pode criar uma procedure pública com a mesma assinatura de OnMouseDown e varrer os componentes do form setando o evento OnMouseDown , se existir, para a sua procedure. Este evento verificará se a tecla ctrl está pressionada, e caso estiver ele chamará o Object Inspector para editar o componente. O evento deverá ser como o da **Listagem 30.** No form principal, no botão "Cobaia", digite o código da **Listagem 31**.

Faça testes, mude cores, tamanhos e posições de componentes, salve, mude outros componentes, veja que Button's e Label's têm Caption enquanto Edit's tem Text. Veja que nunca aparece uma propriedade que o componente não tem.

Listagem 30. Método MouseDown "Geral"

```
procedure TfrmCobaia.Edit1MouseDown(Sender: TObject;
  Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if ssCtrl in Shift then
    AbrirComponente(sender as TComponent);
end;
```

Listagem 31. Chamando a cobaia

```
procedure TfrmTesteRTTI.btCobaiaClick(Sender: TObject);
begin
  // uses uCobaia
  with TfrmCobaia.Create(nil) do
  try
    ShowModal;
  finally
    Release;
  end;
end;
```

Neste ponto é necessário dizer que, se você criar muitas instâncias do mesmo formulário, o Delphi automaticamente muda a propriedade Name das próximas instâncias, concatenando ao nome um sufixo _1, _2 e assim por diante. Isso no nosso banco de dados age como se fosse uma form diferente, fazendo com que você possa ter múltiplas configurações para uma mesma form dependendo do número da instância. Para evitar isso, retire do banco de dados e dos filtros dos ADODataset's o campo "Nome", e trabalhe apenas com o campo "Classe" dos formulários.

Vamos agora fazer um outro teste. Vamos adicionar na form um CheckBox, definir seu evento onMouseDown como sendo o mesmo evento que todos os outros componentes e vamos executar o programa novamente. Como você pode ver todos os componentes mantiveram a configuração que você definiu e agora você pode alterar o novo CheckBox. Faça outro teste diferente. Adicione no banco de dados uma nova propriedade. Lembrando que no nosso script da **Listagem 21** nós adicionamos sete propriedades direto no banco de dados, execute este comando para adicionar a propriedade "Enabled":

```
insert into Propriedades values ('Enabled', 'Boolean');
```

Verifique agora, rodando o programa novamente, que a propriedade Enabled aparece para todos os componentes. E você pode desabilitá-los, mas não pode habilitar depois, porque não pode clicar num componente desabilitado, afinal estamos usando o evento OnMouseDown. Se fizer isso com a propriedade Visible, só encontrará o componente "Desaparecido" no banco e dados. Terá de deletar o registro de sua propriedade Visible para ele reaparecer.

Como não construímos um editor especial para propriedades do tipo boolean igual fizemos com o tipo TColor, temos que escrever os valores "True" e "False", como strings mesmo. O SetPropValue também aceita string e depois internamente converte para o tipo apropriado. Esses dois métodos utilizam internamente um case para tomar uma ação diferente dependendo do tipo da variável. Você pode ver o fonte desses métodos na unit TypInfo e criar o seu próprio, baseado nestes.

Chamando um método pelo seu nome

Com uma instância de um objeto, imagine se você pudesse verificar se um método existe, e então chamar um método pelo seu nome, usando uma string que vem de um banco de dados, por exemplo? Isso seria útil para cálculos de impostos, que são diferentes dependendo do ramo do seu cliente, e cada caso é um caso. Se o seu sistema está flexível o suficiente para criar um objeto de cálculo diferente para cada cliente, mas os objetos não seguem a mesma interface, você poderia criar uma "Formula" para que de alguma maneira pudesse efetuar o cálculo executando uma série de métodos diferentes em ordem diferente. A lista dos métodos viria de um arquivo de texto, ini, XML ou banco de dados. Isso é possível. Abra a nossa classe TCliente na unit uCliente e adicione na seção published o método SayHello, que dispensa uma listagem. Ele apenas faz:

```
MessageBox('Hello World');
```

Crie no formulário principal, na seção type, o seguinte tipo de dado:

```
TNossaProcedure = procedure of object;
```

Isso criará um tipo de dado que é uma referência a um método. No nosso caso um método sem parâmetros. Todo tipo de método pode ser chamado através de uma referência indireta ou um ponteiro, mas métodos published podem ser chamados pelo seu nome, pois tem informações de RTTI. Um evento também nada mais é do que uma referência indireta a um método. Ou seja, um ponteiro vazio para um método que não existe. Quando você edita um evento, você na verdade cria um método e atribui esse método ao ponteiro vazio que é o evento. O evento passa a SER um método.

Crie também no formulário principal uma procedure que aceite um objeto e um segundo argumento do tipo string. Ela será responsável por chamar um método de TCliente pelo seu nome. Criaremos uma variável chamada DadosMetodo que é do tipo TMethod. Esse tipo de dado é uma estrutura, ou Record, que armazena duas informações: um ponteiro para onde começa as instruções do método e outro ponteiro para o objeto que possui o método. Criaremos também a variável CorpoMetodo do tipo TNossaProcedure. DadosMetodo.Data é o ponteiro para o objeto, por isso atribuiremos a ele Pointer(objeto). Pointer converte um ponteiro definido para um ponteiro genérico. DadosMetodo.Code deverá ser um ponteiro para o método em si. Como ponteiros são endereços, a RTTI nos provê a função MethodAddress(nome). Esse método nos retorna o endereço de memória de um método de um objeto cujo nome seja "nome". Se os dois ponteiros do DadosMetodo forem definidos podemos simplesmente converter o DadosMetodo, que é um método genérico, TMethod, para um TNossaProcedure, definida por nós como um método específico, uma procedure of object. Siga o exemplo da **Listagem 32.** Os comentários explicam cada linha.

Crie agora mais um botão no form principal apenas para executar essa procedure que acabamos de criar, passando um TCliente já criado como primeiro parâmetro. Note que o método SayHello de TCliente é executado pelo nome, através de uma string, sem que nem mesmo saibamos se o método realmente existe. O código deste botão você pode conferir na **Listagem 33.**

Listagem 32. MetodoPorNome

```
procedure TfrmTesteRTTI.MetodoPorNome(objeto: Tobject; nome:
string);
var
  //uma estrutura de dados formada por dois ponteiros, uma para
  o objeto e um para o método
  DadosMetodo: TMethod;
  //variável do tipo método para podermos chamá-lo
  CorpoMetodo: TNossaProcedure;
begin
  //objeto convertido para ponteiro genérico é atribuido
  DadosMetodo.Data := Pointer(objeto);
  //capturamos o endereço do método por nome
  DadosMetodo.Code := objeto.MethodAddress(nome);
  //se não existir saimos sem fazer nada
  if not Assigned(DadosMetodo.Code) then
    Exit;
  //atribuimos nossa estrutura TMethod, formada por ponteiros
  para métodos, à nossa variável do tipo método
  CorpoMetodo := TNossaProcedure(DadosMetodo);
  //chamamos o método
  CorpoMetodo;
end;
```

Listagem 33. Executando MetodoPorNome

```
procedure TfrmTesteRTTI.btPorNomeClick(Sender: TObject);
var
  obj: TCliente;
begin

  obj := TCliente.Create;
  try
    MetodoPorNome(obj, 'SayHello');
  finally
    obj.Free;
  end;

end;
```

Na **Listagem 34** você pode conferir uma versão simplificada da procedure MetodoPorNome. Pulamos uma etapa e estamos executando o DadosMetodo como um TNossaProcedure direto na hora da conversão.

Listagem 34. MetodoPorNome simplificado

```
procedure TfrmTesteRTTI.MetodoPorNome(objeto: Tobject; nome:
string);
var
  DadosMetodo: TMethod;
begin
  DadosMetodo.Data := Pointer(objeto);
  DadosMetodo.Code := objeto.MethodAddress(nome);
  if not Assigned(DadosMetodo.Code) then
  Exit;
  TNossaProcedure(DadosMetodo)();
end;
```

Estude entre as linhas 200 e 220 da unit TypInfo para ter um vislumbre do que pode fazer com essas informações. Todos os eventos dos componentes e métodos published podem ser capturados usando o método GetMethodProp desta unit. Essas rotinas podem ser chamadas por nome e até trocadas entre si, em tempo de execução, se tiverem a mesma assinatura. Espero que tire bastante proveito destas informações e crie sua própria IDE de relatórios!

Conclusão

Neste artigo vimos como usar vários segredos da RTTI na prática. Não tivemos a pretensão de elaborar uma documentação

extensa e completa da RTTI, mas tivemos um vislumbre do que é possível criar com ela. Usando a RTTI podemos criar frameworks de produtividade, clonar objetos, criar programas mais flexíveis, desacoplados e customizáveis. Associadas a padrões de projeto e boas práticas de programação, estas técnicas podem conduzir a sistemas bastante genéricos, que podem ser instalados em uma gama maior de empresas e configurados pelos próprios usuários, reduzindo os custos de manutenção e suporte. Usando esses conhecimentos mais profundamente você pode criar repositórios de classes para que todos os seus softwares se baseiem nesses padrões, e também pode criar métodos para que os usuários criem suas próprias janelas de cadastro e relatórios.

Como última dica, recomendo que o leitor estude alguns frameworks já existentes para Delphi, que também usam RTTI internamente, como por exemplo o Infra, DePo, Jazz e InstantObjects, para facilitar o seu caminho rumo a uma aplicação 100% POO, desacoplada, bugfree, de baixo custo de implementação e manutenção e alta lucratividade. ●

Links

SQL Server Express

http://www.microsoft.com/downloadS/details.aspx?familyid=220549B5-0B07-4448-8848-DCC397514B41&displaylang=en

MS SQL Server Management Studio Express

http://www.microsoft.com/downloadS/details.aspx?familyid=C243A5AE-4BD1-4E3D-94B8-5A0F62BF7796&displaylang=en

NHibernate

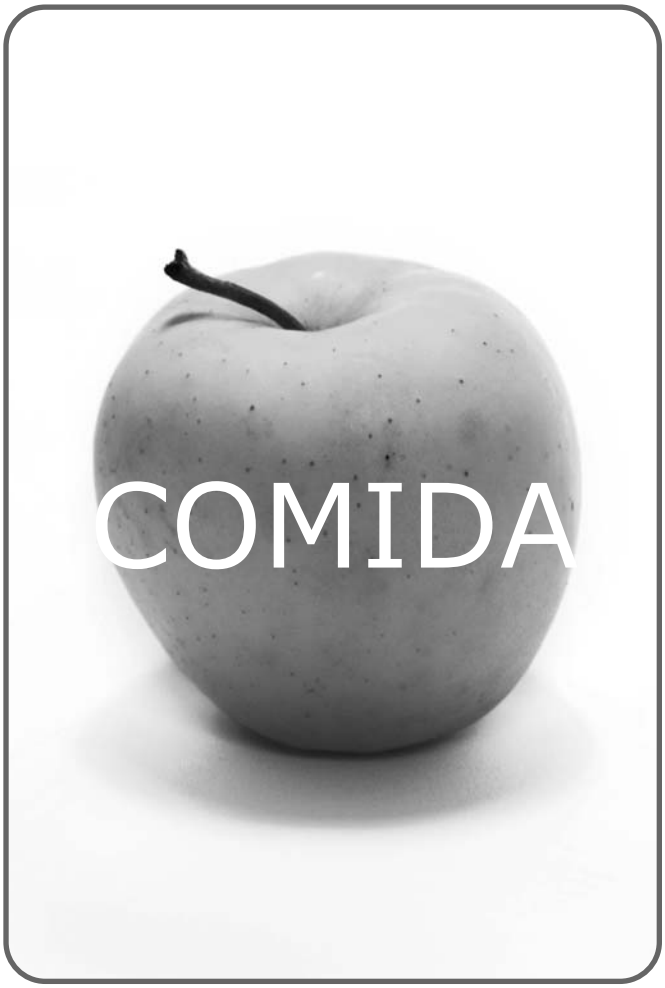
http://www.nhforge.org

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback



Existem coisas que não conseguimos ficar sem!

...só pra lembrar, sua assinatura pode estar acabando!

Renove Já!

www.devmedia.com.br/renovacao

Para mais informações:
www.devmedia.com.br/central

DevMedia GROUP

Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

Padrão Command

O Senhor Encapsulamento

Do que trata o artigo
O artigo aborda alguns dos tipos de encapsulamento e apresenta o Padrão Command como uma solução para criar um sistema robusto e flexível. Para exemplificar o uso deste padrão é criado um exemplo de como implementar o recurso Desfazer em um programa.

Para que serve
Técnicas de encapsulamento ajudam você a criar um sistema fechado contra erros e ao mesmo tempo aberto para expansão. Haverá uma integração maior entre as partes do programa, pois o encapsulamento juntamente com o Padrão Command permite que estas partes se comuniquem sem saber o que realmente cada uma faz.

Em que situação o tema é útil
Usando esta técnica em conjunto com o Padrão Command você ganha um "cinto de utilidades" cheio de novos recursos interessantes para implantar no sistema. Um deles é o recurso de desfazer operação apresentado neste artigo.

Resumo do DevMan

Saber como aproveitar ao máximo o que o encapsulamento tem para oferecer possibilita a você criar grandes sistemas garantindo a facilidade de manutenção durante a vida do mesmo. Com um pequeno, exemplo disponível para download, você vai aprimorar um pouco mais seus conhecimentos sobre esta técnica e ainda vai aprender como implementar o Padrão Command. Este padrão traz consigo um conjunto de recursos muito interessantes, como desfazer e refazer operações apresentado neste artigo, além de permitir que você faça um controle detalhado de tudo que seu programa executar.

Rafael Stavarengo
faelsta@gmail.com
Programador-analista da Cheina Informática. Trabalha com Delphi, JME e PHP + ZendFramework. Formando em Tecnologia de Análise e Desenvolvimento de Sistemas pela UNIPAR. Blog: http://www.devmedia.com.br/stavarengo

Neste artigo você vai aprender na prática como funciona o Padrão Command. Este padrão possibilita que partes do programa sejam encapsuladas em pequenas classes de comandos que serão executadas através de um invocador de comandos. Durante o artigo será criado um projeto de

exemplo, para simular um sistema onde serão criadas algumas classes de comandos para fazer um cadastro qualquer. Após mostrar como são as classes de comando, é explicado como usar o Padrão Command para adicionar o recurso de desfazer operações, que você poderá facilmente modificar para que também seja possível refazer algo que havia sido desfeito. No projeto de exemplo trabalharei apenas com o código da aplicação para tornar o artigo menos extenso, ou seja, tudo será feito sem usar um único formulário sequer. Mas no arquivo disponível para download você vai encontrar o mesmo exemplo, porém com formulários como qualquer aplicação.

Encapsulamento

Encapsulamento é um dos fundamentos básicos da programação orientada a objetos. É o ato de proteger algo, normalmente atributos de uma classe, para que outras classes não acessem diretamente este atributo. Encapsular, segundo dicionário Michaelis significa: Incluir, proteger em uma cápsula ou como em uma cápsula. Na **Listagem 1** existe um exemplo de uma classe cujo os atributos NÃO ESTÃO encapsulados.

Listagem 1. Classe TCliente sem encapsulamento

```
1  unit Cliente;
2  Interface
3  Type
4    TCliente = class
5    Private
6      FCpf: string;
7      FNome: string;
8    Published
9      property Nome: string read FNome write FNome;
10      property Cpf: string read FCpf write FCpf;
11   end;
12 Implementation
13 end.
```

Imagine uma classe que represente a entidade cliente (**Listagem 1**): ela poderia receber o nome TCliente e conforme o exemplo possui dois atributos (propriedades): Nome e CPF. Olhando com mais atenção nas linhas 9 e 10 você verá que quando os valores destas propriedades são lidos (read) e escritos (write), é feito um acesso direto às variáveis onde os valores serão armazenados. Isto significa que estas propriedades não estão encapsuladas, pois, apesar das variáveis (linhas 6 e 7) estarem na sessão private, o usuário pode alterar estes valores através das propriedades declaradas nas linhas 9 e 10. Do jeito como está, a classe TCliente poderá aceitar números de CPF inválido, pois nenhuma validação será feita.

Para garantir que um usuário nunca irá conseguir salvar um número inválido de CPF é preciso encapsular (empacotar, proteger), o acesso à variável declarada na linha 6. Para conseguir isto vou criar dois novos métodos: SetCpf e GetCpf. Veja **Listagem 2**.

Nada de surpreendente. Apenas criei dois novos métodos: GetCpf e SetCpf que podem ser vistos nas linhas 8 e 9 da **Listagem 2**. Observe na linha 12 desta mesma listagem que os valores de acesso (write) e de leitura (read) da propriedade CPF foram alterados. Desta maneira, sempre que o CPF do cliente for alterado ele vai passar obrigatoriamente pelo método SetCpf. O mesmo acontece para pegar o CPF atual, pois para isso será obrigatório passar pelo método GetCpf. O que nos interessa neste caso é o método SetCpf, pois nele estão definidas as condições para que um CPF seja aceito. Veja entre as linhas 21 e 25 que a única regra é o CPF ter exatamente 11 caracteres.

Listagem 2. Encapsulando a variável FCpf

```
1  unit Cliente;
2  Interface
3  Type
4    TCliente = class
5    Private
6    FCpf: string;
7    FNome: string;
8    function GetCpf: string;
9    procedure SetCpf(const Value: string);
10   Published
11      property Nome: string read FNome write FNome;
12      property Cpf: string read GetCpf write SetCpf;
13   end;
14 Implementation
15 uses SysUtils;
16  { TCliente }
17 function TCliente.GetCpf: string;
18 Begin
19 Result := FCpf;
20  end;
21  procedure TCliente.SetCpf(const Value: string);
22  Begin
23    if Length(Value) <> 11 then
24      raise Exception.Create
        ('O CPF precisar ter 11 caracteres');
25    FCpf := Value;
26  end;
27  end.
```

Quando criamos estes dois métodos estamos reforçando o encapsulamento. Um dos objetivos do encapsulamento é proteger algo. Na classe TCliente, quando criei o método de acesso SetCpf para a variável FCpf, também criei uma oportunidade de controlar o que pode ou não ser colocado na variável, já que para qualquer outra classe acessar esta variável será obrigado executar o método SetCpf. Entendido este raciocínio, que tal deixar o código um pouco mais simples de ler. Veja o que pode ser feito (**Listagem 3**).

O funcionamento da classe TCliente não mudou em nada entre as **Listagens 2** e **3**. Porém a **Listagem 3** está seguindo um padrão de codificação mais simples: as propriedades declaradas na sessão published foram removidas, pois esta sessão não será útil para nós. O que restou foram os métodos de "empacotamento" get e set. Veja entre as linhas 9 e 12 que estes métodos foram movidos para a sessão public da classe. As variáveis FCpf e FNome continuam escondidas na sessão private e não podem ser tiradas de lá para não ferir o encapsulamento. Colocar estas variáveis em outra sessão, como por exemplo a sessão public, faria com que qualquer outra classe pudesse acessar estas variáveis diretamente e assim não faria sentido a existência dos métodos de encapsulamento get e set.

A partir de agora o encapsulamento será levado para um nível mais alto: vamos encapsular a chamada de métodos. Ao encapsular métodos, você "cristaliza" partes do seu programa para que os objetos possam usar estas funções (as partes), sem saber como as coisas são feitas. Mas, o que isto significa? Na prática, isto quer dizer que você pode pegar determinadas operações que seu programa executa, como

por exemplo, salvar um registro ou imprimir um relatório e colocá-las dentro de um único lugar que terá sua chamada encapsulada por outra classe. Encapsular a chamada de métodos também permite que você faça coisas engenhosas como registrar as chamadas para fins de registro ou disponibilizar o recurso de desfazer, entre outras coisas descritas no decorrer do artigo.

Listagem 3. Adotando um padrão de codificação

```
1  unit Cliente;
2  Interface
3  Type
4    TCliente = class
5    Private
6      FCpf: string;
7      FNome: string;
8    Public
9      function GetNome: string;
10     procedure SetNome(const Value: string);
11     function GetCpf: string;
12     procedure SetCpf(const Value: string);
13   end;
14 Implementation
15 uses SysUtils;
16 { TCliente }
17 function TCliente.GetCpf: string;
18 Begin
19   Result := FCpf;
20 end;
21 function TCliente.GetNome: string;
22 Begin
23   Result := FNome;
24 end;
25 procedure TCliente.SetCpf(const Value: string);
26 Begin
27   if Length(Value) <> 11 then
28     raise Exception.Create
         ('O CPF precisar ter 11 caracteres');
29   FCpf := Value;
30 end;
31 procedure TCliente.SetNome(const Value: string);
32 Begin
33   FNome := Value;
34 end;
35 end.
```

Vamos encapsular!

Veja agora como encapsular partes do programas em classes. No Delphi, crie um novo projeto e feche a unit do formulário principal sem salvar, não vamos precisar de formulários por enquanto. Peça uma nova unit (menu File>New>Unit), e escreva o código da **Listagem 4** – não deixe de ler os comentários desta e das outras listagens, eles irão ajudar você a entender o código.

A **Listagem 4** apresenta uma classe bem simples. Ela apenas retorna as informações de conexão. É como se fosse uma classe de clientes, que serviria para representar a tabela de clientes do banco de dados. Olhe de novo para ela e você verá que a classe TConfiguracoes não grava as informações em lugar algum. Ela apenas retorna os valores das variáveis declaradas nas linhas 9, 10 e 11 e para alterar os valores destas variáveis é necessário executar um dos métodos de encapsulamento (de atributos) get, que estão declarados nas linhas 16, 24 e 32.

Se esta classe não sabe onde os valores de configurações são salvos, então quem sabe? Os comandos é a resposta. Neste nosso exemplo vou encapsular o gerenciamento das informações de conexão. Quando digo gerenciamento, quero dizer as funções de incluir, localizar, atualizar e excluir. Vou começar pela função de salvar. Para encapsular uma função do seu sistema, seja ela qual for, crie uma unit com um código parecido com o da **Listagem 5**.

Nota do DevMan

Membros da sessão published têm a mesma visibilidade que os membros da sessão public com a diferença de que, até a versão 2009 do Delphi, as informações de tempo de execução (RTTI) são geradas apenas para membros da sessão published. Além disto, published possui alguma restrição sobre os tipos dos membros declarados. Alguns tipos simplesmente não são permitidos.

Listagem 4. Classe TConfiguracoes

```
1  unit Configuracoes;
2  {**
3   * Esta classe representa as configurações de conexão do sistema.
4   *}
5  interface
6  Type
7    TConfiguracoes = class
8    private
9      CaminhoDoBanco: string;
10     Usuario: string;
11     Senha: string;
12   public
13     {**
14      * Retorna a string para conexão com o banco de dados
15      *}
16     function GetCaminhoDeConexao: string;
17     {**
18      * Define o caminho de conexão com o banco de dados
19      *}
20     procedure SetCaminhoDeConexao(Caminho: string);
21     {**
22      * Retorna o nome de usuário para logar no banco de dados
23      *}
24     function GetUsuario: string;
25     {**
26      * Define o nome de usuário para logar no banco de dados
27      *}
28     procedure SetUsuario(Usuario: string);
29     {**
30      * Retorna a senha para conexão no banco de dados
31      *}
32      function GetSenha: string;
33     {**
34     * Define a senha para conexão com o banco de dados
35     *}
36     procedure SetSenha(Senha: string);
37   end;
38 implementation
39 { TConfiguracoes }
40 function TConfiguracoes.GetCaminhoDeConexao: string;
41 Begin
42   Result := CaminhoDoBanco;
43 end;
44 function TConfiguracoes.GetSenha: string;
45 Begin
46   Result := Senha;
47 end;
48 function TConfiguracoes.GetUsuario: string;
49 Begin
50   Result := Usuario;
51 end;
52 procedure TConfiguracoes.SetCaminhoDeConexao(Caminho: string);
53 Begin
54   CaminhoDoBanco := Caminho;
55 end;
56 procedure TConfiguracoes.SetSenha(Senha: string);
57 Begin
58   Self.Senha := Senha;
59 end;
60 procedure TConfiguracoes.SetUsuario(Usuario: string);
61 Begin
62   Self.Usuario := Usuario;
63 end;
64 end.
```

Nota do DevMan

O gerenciamento de informações de um cadastro – salvar, buscar, atualizar, excluir – também é conhecido como CRUD. CRUD é um acrônimo da uma expressão inglesa, cujas letras significam: Create (salvar, criar), Retrieve/Read (buscar, ler), Update (atualizar) e Delete (excluir).

Listagem 5. Uma função encapsulada – um comando

```
1  unit SalvarConfiguracoesCommand;
2  Interface
3  uses Configuracoes;
4  Type
5    TSalvarConfiguracoesCommand = class
6    Private
7      ConfiguracoesParaSalvar: TConfiguracoes;
8    Public
9      constructor Create(ConfiguracoesParaSalvar: TConfiguracoes);
10      procedure Execute;
11   end;
12 Implementation
13 uses Registry, Windows, SysUtils;
14  { TSalvarConfiguracoesCommand }
15  constructor TSalvarConfiguracoesCommand.Create(
16    ConfiguracoesParaSalvar: TConfiguracoes);
17  Begin
18    inherited Create;
19    Self.ConfiguracoesParaSalvar := ConfiguracoesParaSalvar;
20  end;
21  procedure TSalvarConfiguracoesCommand.Execute;
22   Var
23     Registro: TRegistry;
24   Begin
25    {**
26     * As próximas três linhas poderiam ser apenas uma,
27     * se também fosse usado o padrão Factory.
28     *}
29    Registro := TRegistry.Create;
30    Registro.RootKey := HKEY_LOCAL_MACHINE;
31    Registro.OpenKey('Software\Padrão Command', True);
32    Registro.WriteString('Caminho', ConfiguracoesParaSalvar.
       GetCaminhoDeConexao);
33    Registro.WriteString('Usuario', ConfiguracoesParaSalvar.
       GetUsuario);
34    Registro.WriteString('Senha', ConfiguracoesParaSalvar.
       GetSenha);
35    Registro.CloseKey;
36    FreeAndNil(Registro);
37  end;
38  end.
```

A **Listagem 5** apresenta outra classe muito simples em termos de codificação. Na classe TSalvarConfiguracoesCommand existem apenas dois métodos (linhas 9 e 10), sendo que um deles é o próprio construtor. Pois bem, este construtor é bem interessante. Ele obriga que um objeto do tipo TConfiguracoes (**Listagem 4**), seja criado e passado como parâmetro. Isto significa que para criar uma classe TSalvarConfiguracoesCommand, você precisa informar um objeto do tipo TConfiguracoes. Internamente a classe TSalvarConfiguracoesCommand armazena uma cópia do objeto recebido para usá-lo quando o método Execute for chamado.

Para testar este código, faça o seguinte: abra o código fonte do arquivo DPR do seu projeto, através do menu Project > View Source e escreva o código conforme a **Listagem 6**.

O que nos interessa está nas linhas 8, 11, 12, 13, 14, e 15. Na linha 8, eu declarei uma variável do tipo TConfiguracoes para definir os valores de conexão que desejo salvar. Entre as linhas 11 e 14, instanciei a variável e defini os valores da conexão. O mais importante acontece na linha 15, eu peço para classe TSalvarConfiguracoesCommand salvar as configurações que foram definidas. Tudo muito simples, porém com consequências boas de grande impacto. Estamos ainda no meio do caminho para compreender o Padrão Command, contudo já alcançamos um bom resultado no que se diz respeito à manutenção do sistema, pois o código para salvar as configurações está em um único lugar. Melhor do que isto, ninguém mais além da própria classe TSalvarConfiguracoesCommand sabe como e onde estas informações são salvas. Isto significa que se estas informações fossem salvas em um banco de dados, seria possível trocar o banco de dados da aplicação sem precisar mudar uma linha que não seja dentro da classe TSalvarConfiguracoesCommand. Se você achou esquisito o código da linha 15 na **Listagem 6**, vai gostar de saber que aquela linha poderia ter sido escrita em pelo menos mais duas formas diferentes. Veja **Listagem 7**.

Listagem 6. Testando a classe TSalvarConfiguracoesCommand

```
1  program Project1;
2  Uses
3    Forms,
4    Configuracoes in 'Configuracoes.pas',
5    SalvarConfiguracoesCommand in 'SalvarConfiguracoesCommand.pas';
6  {$R *.res}
7  Var
8     Config: TConfiguracoes;
9  Begin
10    Application.Initialize;
11    Config := TConfiguracoes.Create;
12    Config.SetCaminhoDeConexao('Caminho para conexão');
13    Config.SetUsuario('SYSDBA');
14    Config.SetSenha('masterkey');
15    TSalvarConfiguracoesCommand.Create(Config).Execute;
16    Application.MessageBox('As configurações foram salvas',
       'Aviso', 48);
17    Application.Run;
18 end.
```

Listagem 7. Outras maneiras de escrever o código da listagem 6

```
1  {**
2   * Primeira alternativa
3   *}
4  Var
5    Config: TConfiguracoes;
6    Comando: TSalvarConfiguracoesCommand;
7  Begin
8    Comando := TSalvarConfiguracoesCommand.Create(Config);
9    Comando.Execute;
10    FreeAndNil(Comando);
11 {**
12  * Segunda alternativa
13  *}
14   with TSalvarConfiguracoesCommand.Create(Config) do
15   Begin
16     Execute;
17  end;
```

O Padrão Command: Os personagens envolvidos

Uma definição para o padrão Command seria: Encapsula uma solicitação como se fosse um objeto, permitindo que você parametrize clientes com diferentes solicitações, enfileire ou registre solicitações e implemente operações reversíveis.

O que você viu até agora foi a ideia de como encapsular funções do sistema em pequenas classes independentes, cuja única razão de existirem é o fato de saberem executar (procedure Execute) alguma operação do sistema – mas isto ainda não chega perto do que você pode fazer com o Padrão Command. O que existe até agora é uma classe que executa várias outras operações internamente para atingir um objetivo – neste caso

o objetivo é salvar a configurações de conexão. Esta classe também poderia invocar outros comandos internamente. Não há regras quanto ao que uma classe pode ou não fazer. O conceito é único: as classes devem ser capazes de executar a operação que se propõe quando o método Execute for invocado.

Olhe para as classes que criamos até agora e tente responder: como você poderia criar um registro dos comandos que são executados para que fosse possível desfazê-los posteriormente? Tente encontrar uma solução para isto. Se você pensou e chegou à conclusão de que está faltando algo para isto ser possível você acertou. Até agora o Padrão Command ainda não apareceu, então o que está faltando? Vejamos a resposta a seguir.

Para poder afirmar que o Padrão Command foi implementado é preciso de no mínimo três coisas. A primeira delas é uma classe que servirá apenas para invocar os comandos existentes (esta classe é chamada de classe invocadora), depois deve existir uma interface para as classes que desejam ser uma classe de comando e por último, é claro que deveria existir um grupo de outras classes para implementar a interface de comando. As classes deste grupo serão o que chamo de classe de comando. Veja como fica o design deste padrão na **Figura 1**.

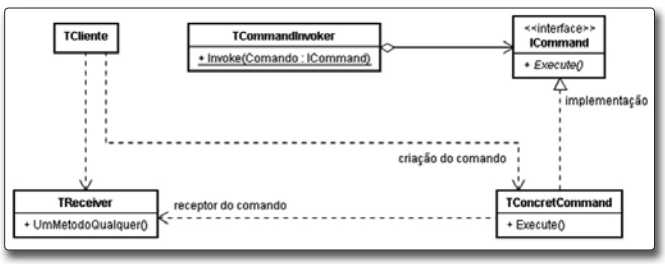


Figura 1. Diagrama oficial do Padrão Command

O diagrama da **Figura 1** exibe o mínimo de personagens necessários para existir o Padrão Command. Entenda cada classe deste diagrama como um personagem dentro do padrão. Não necessariamente existirá um diagrama igualzinho a este no seu projeto. Veja o que significa cada personagem do diagrama:

TCliente – é o responsável por instanciar o objeto de uma classe de comando. Ou seja, TCliente pode ser qualquer classe no seu sistema, inclusive outras classes de comandos;

TCommandInvoker – é a classe que invoca os comandos. Esta é a principal diferença entre o Padrão Command e o que você viu nas listagens até agora. Volte à **Listagem 6** e veja na linha 15 que é o próprio arquivo DPR quem cria e executa o comando para salvar as configurações. Neste caso o arquivo DPR seria o que representa a classe TCliente no diagrama. Porém para isto se tornar uma realidade as conformidades do padrão devem ser seguidas: o cliente – neste caso o arquivo DPR, não pode fazer a chamada do método execute. Para isto existe a classe invocadora: TCommandInvoker;

TConcreteCommand – esta classe representa o papel que seria da classe TSalvarConfiguracoesCommand, ou seja, é o comando propriamente dito. Mas porque não posso dizer que a classe TSalvarConfiguracoesCommand é um comando? Realmente você não pode dizer isto pois para afirmar que uma classe é um comando é necessário que ela implemente uma interface de comando. É aí que entra a interface ICommand;

ICommand – interface para todas as classes de comando. Observe na **Figura 1** que o único método da classe TCommandInvoker recebe como parâmetro um objeto de qualquer classe, desde que esta implemente a interface ICommand. O invocador não irá aceitar TSalvarConfiguracoesCommand enquanto a interface ICommand não for implementada por ela. Este é um dos motivos para o qual você não pode chamar a classe TSalvarConfiguracoesCommand de classe de comando, pois ela não implementa uma interface de comando;

TReceiver – receptor é o personagem do diagrama que sabe como executar as tarefas para que a solicitação seja executada. Qualquer classe pode atuar como um receptor – se a classe TSalvarConfiguracoesCommand fosse uma classe de comando, então TConfiguracoes e TRegistry seriam as classes receptoras que o diagrama se refere como TReceiver;

Implementando o padrão

Seguindo o que o diagrama da **Figura 1** diz, é necessário criar uma interface para as classes de comando e outra classe que será a invocadora destes comandos. Veja as **Listagens 8 e 9**.

Listagem 8. Classe invocadora de comandos (TCommandInvoker)

```
1  unit CommandInvoker;
2  Interface
3  uses Command;
4  Type
5    TCommandInvoker = class
6    Public
7      class procedure Invoke(Comando: ICommand);
8    end;
9  Implementation
10 { TCommandInvoker }
11 class procedure TCommandInvoker.Invoke(Comando: ICommand);
12 Begin
13   Comando.Execute;
14 end;
15 end.
```

Não fique surpreso. A classe invocadora de comandos – **Listagem 8** – é realmente simples. Como já foi dito, a responsabilidade dela é invocar os comandos. Ela faz isto chamando o método Execute. Em outras palavras, a classe TCommandInvoker é responsável por encapsular a chamada do método Execute das classes de comando. Você pode estar perguntando: como a classe TCommandInvoker pode garantir que os objetos passados como parâmetro no método Invoke possuem o método Execute? A resposta é a seguinte: ela pode garantir isto pois o método Invoke exige que o objeto passado como parâmetro seja de alguma classe que implemente a interface ICommand. Por sua vez a interface ICommand obriga as classes que a implementam declarar o método Execute (veja linha 6 da **Listagem 9**). É assim que a classe TCommandInvoker consegue garantir que os objetos recebidos no parâmetro Comando do método Invoke possuem o método Execute.

A **Listagem 9** apresenta a interface que as classes de comando devem implementar. Esta interface também é muito simples: ela obriga a declaração do método Execute (linha 6 da **Listagem 9**).

Com a interface ICommand pronta, que tal implementá-la na classe TSalvarConfiguracoesCommand? Veja como fica na **Listagem 10**.

Listagem 9. Interface para classes de comando

```
1 unit Command;
2 Interface
3 Type
4   ICommand = interface
5   ['{54B4915A-96E0-47BA-AE37-A18347D5B466}']
6   procedure Execute;
7   end;
8 Implementation
9 end.
```

Listagem 10. Implementando a inteface ICommand

```
1 unit SalvarConfiguracoesCommand;
2 Interface
3 uses Configuracoes, Command;
4 Type
5   TSalvarConfiguracoesCommand = class(TInterfacedObject,
    ICommand)
6   Private
7     ConfiguracoesParaSalvar: TConfiguracoes;
8   Public
9     constructor Create(ConfiguracoesParaSalvar:TConfiguracoes);
10     procedure Execute;
11  end;
```

Não coloquei todo o código da classe TSalvarConfiguracoesCommand na **Listagem 10** porque apenas duas linhas sofreram alterações: são as linhas 3 e 5. Na linha 3 adicionei a unit Command que possui a declaração da interface ICommand (veja **Listagem 9**), e na linha 5 mudei a declaração da classe TSalvarConfiguracoesCommand para que ela implemente a interface ICommand. Após estas alterações, se o método Execute da linha 10 for removido ou renomeado a unit não irá compilar, pois qualquer classe que implemente ICommand tem a obrigação de declarar o método Execute exatamente igual à declaração dele na interface ICommand. Somente a partir de agora é que a classe TSalvarConfiguracoesCommand pode ser chamada de classe de comando.

Agora falta apenas mudar a maneira como o comando é invocado no arquivo DPR. Novamente abra o código fonte do arquivo DPR (Project > View Source) e altere o código como na **Listagem 11**.

Listagem 11. Invocando o comando para salvar as configurações

```
1   {$R *.res}
2  Var
3     Config: TConfiguracoes;
4  Begin
5    Application.Initialize;
6    Config := TConfiguracoes.Create;
7    Config.SetCaminhoDeConexao('Caminho para conexão');
8    Config.SetUsuario('SYSDBA');
9    Config.SetSenha('masterkey');
10   TCommandInvoker.Invoke(TSalvarConfiguracoesCommand.
     Create(Config));
11   Application.MessageBox('As configurações foram salvas',
     'Aviso', 48);
12   Application.Run;
13 end.
```

Poucas mudanças na **Listagem 11**. Apenas a linha 10 foi alterada. Agora a chamada do método Execute está encapsulada (protegida), dentro do método Invoke, ao contrário do que acontece na linha 15 da **Listagem 6,** onde a chamada do Execute está exposta.

Nota do DevMan

Você notou a palavra class no início das linhas 7 e 11 da **Listagem 8**? Métodos com esta assinatura são chamados de métodos estáticos. Este tipo de método pode ser chamado diretamente da classe sem a necessidade de instanciar um objeto da classe para chamá-lo. Veja na linha 10 da **Listagem 11** que o método Invoke é chamado sem que seja necessário criar um objeto da classe TCommandInvoker.

Cristalização, um belo nome para o encapsulamento

Agora vem um fato muito interessante. Sabemos que ninguém mais além da classe TCommandInvoker pode invocar comandos. Mais do que isto, é possível afirmar que ninguém mais além da classe TCommandInvoker sabe ao menos o nome do método a ser chamado para executar comandos. Isto é uma forma de encapsulamento. Neste caso foi encapsulada a chamada de métodos. Com o Padrão Command você encapsula a chamada do método Execute dentro do método Invoke, assim como foi encapsulado o acesso ao atributo CPF da classe TCliente dentro dos métodos GetCpf e SetCpf (veja **Listagem 2**). A situação é a mesma. A diferença sutil está no fato de que a classe TCliente encapsula o acesso a um atributo e o Padrão Command encapsula a chamada de um método. Assim como foi adicionada uma regra pra aceitar um número de CPF na classe TCliente (linha 23 da **Listagem 2**), também podemos adicionar regras ou controles dentro do método Invoke para chamar o método Execute dos comandos.

Responda: como você pode executar uma determinada ação sempre que um cliente for excluído, se existirem diversos lugares de onde o usuário possa ser excluído? Fica difícil, pois você terá que procurar e encontrar todos os lugares do programa onde é possível excluir um cliente para adicionar o código da ação desejada. Mesmo assim ainda corre o risco de aparecer um novo lugar. Se isto acontecer você terá a obrigação de lembrar de adicionar uma nova cópia do código para executar a ação desejada.

Agora pense um pouco. Se você criar um algoritmo único para excluir clientes e colocá-lo dentro de uma classe de comando, onde apenas uma única classe chamada TCommandInvoker saiba como invocá-la, seria muito simples executar uma ação quando um cliente fosse excluído, já que para excluir o cliente é necessário invocar o comando de exclusão e para invocar qualquer comando, é necessário passar pelo método Invoke. Sendo assim você vai precisar escrever o código da ação que deseja executar quando um cliente for excluído em um único lugar. Veja como seria na **Listagem 12**.

Ações reversíveis

Talvez você pense que implementar operações reversíveis seja algo difícil, demorado e complicado. Ao utilizar o Padrão

Command tudo fica facilitado. Olhe para as classes criadas até agora e tente imaginar como você poderia implementar operações reversíveis no exemplo criado.

É sabido que todas as classes de comando implementam a mesma interface: ICommand. Também é sabido que uma interface define regras (ou contratos), que as classes devem seguir para poder implementá-la. Foi o que aconteceu com a classe TSalvarConfiguracoesCommand. Após implementar a interface ICommand tornou-se impossível excluir ou renomear o método Execute. Partindo deste princípio, o que acontece se mais um método for adicionado à interface ICommand? Resposta: todas as classes que a implementam também deverão declarar este novo método. Isto é o que vou fazer. Vou declarar uma nova regra para a interface ICommand. Veja **Listagem 13**.

Listagem 12. Controlando a execução dos comandos

```
1  class procedure TCommandInvoker.Invoke(Comando: ICommand);
2   Begin
3    if Comando is TExcluirClienteCommand then
4      Log.Add('Um cliente foi excluído');
5    Comando.Execute;
6  end;
```

Listagem 13. Uma nova regra para os comandos

```
1 unit Command;
2 Interface
3 Type
4    ICommand = interface
5      ['{54B4915A-96E0-47BA-AE37-A18347D5B466}']
6      procedure Execute;
7      procedure Undo;
8    end;
9 Implementation
10 end.
```

Nota

A **Listagem 8** apresenta um exemplo com objetivo didático. Você deve evitar se prender ao nome das classes que você cria, como aparece na linha 3 da **Listagem 12**. Isto traria futuros transtornos. As boas práticas de programação abominam (mas não excluem), situações onde é necessário escrever o nome de alguma classe no algoritmo. Você deve evitar isto com todas as suas forças.

Se você notou a diferença das **Listagens 9** e **13**, então já deve ter matado a charada sobre a implementação do recurso desfazer. Caso tenha alguma ideia de como será implementado este recurso, seja lá qual for ela, este é o momento de tentar colocá-la em prática antes de continuar a leitura, assim será mais fácil compreender o que vem a seguir.

A única diferença entre as listagens está na linha 7 da **Listagem 13**. Uma nova regra para todas as classes que desejam ser consideradas classes de comando foi criada. A partir de agora elas devem declarar dois métodos: Execute e Undo. Neste momento não é possível compilar o projeto pois o seguinte erro de compilação irá aparecer na unit SalvarConfiguracoesCommand.pas: [Error] SalvarConfiguracoesCommand.pas(14): Undeclared identifier: 'Undo'. Este erro está dizendo que para a classe TSalvarConfiguracoesCommand continuar a ser considerada uma classe de Command, o método Undo deve ser implementado. É isto que você pode ver na **Listagem 14**.

Listagem 14. Declarando o método Undo

```
1  unit SalvarConfiguracoesCommand;
2  Interface
3  uses Configuracoes;
4  Type
5    TSalvarConfiguracoesCommand = class
6    Private
7      ConfiguracoesParaSalvar: TConfiguracoes;
8    Public
9      constructor Create(ConfiguracoesParaSalvar:
        TConfiguracoes);
10     procedure Execute;
11     procedure Undo;
12   end;
13 Implementation
14  uses Registry, Windows, SysUtils;
15  { TSalvarConfiguracoesCommand }
16 //****Aqui vai a implementação do construtor****\\
17 //****Aqui vai a implementação do método Execute****\\
18  procedure TSalvarConfiguracoesCommand.Undo;
19  Begin
20    // TODO: Este método precisa ser implementado
21 end;
22 end.
```

Na **Listagem 14** eu declarei o novo método que a interface ICommand está pedindo (linhas 11, 18, 19, 20, 21), porém não fiz a implementação do mesmo, ou seja, se o método Undo fosse chamado agora nada iria acontecer. Fiz isto porque antes de implementar o método de desfazer é necessário definir como as ações serão desfeitas.

Ações reversíveis – uma ideia nascendo

Imagine esta situação do dia-a-dia: o usuário altera um registro de uma tabela qualquer. Neste caso, o que é necessário ter em mãos para desfazer alteração feita no registro? Todos devem ter pensado a mesma coisa: para desfazer a alteração de um registro é necessário ter os valores do registro antes dele ser alterado. É mais ou menos assim, existe uma tabela onde é armazenado o nome e o telefone de uma pessoa; nesta tabela existe o seguinte registro: Erich Gamma – 555-3636; o usuário altera este registro para Richard Helm – 555-3636; após algum tempo ele pressiona o botão desfazer e o registro deve voltar a ser Erich Gamma – 555-3636. No exemplo deste artigo, o usuário pode desfazer as alterações que ele fizer nas configurações de conexão do sistema. Veja o passo a passo de como uma ação será desfeita no diagrama de comunicação da **Figura 2**.

O diagrama de comunicação apresentado na **Figura 2** expõe todos os passos executados para que um comando invocado possa ser desfeito. Entenda o que este diagrama está mostrando, nos passos a seguir.

A. Passo 1 – O usuário clica no botão Salvar para salvar uma determinada configuração de conexão;

B. O evento OnClick do botão que ele clicou cria um objeto da classe TSalvarConfiguracoesCommand em uma variável chamada ComandoSalvar (passo 2);

C. Ainda no evento OnClick, uma chamada ao método Invoke da classe TCommandInvoker é feita (passo 3). Como parâmetro é passado o objeto ComandoSalvar criado no passo 2;

D. Por sua vez, a classe invocadora chama o método Execute do objeto de comando. Veja passo 4;

E. Dentro do método Execute da classe TSalvarConfiguracoesCommand, é feita uma busca pela configuração de conexão

atual antes de salvar a nova configuração. Isto é feito para que seja possível restaurar estas configurações mais tarde, quando o método Undo for chamado. Para buscar a cofinguração existente, TSalvarConfiguracoesCommand invoca (passo 5 e 6) o comando TBuscarConfiguracoesComand – ainda não mostrei como esta classe funciona, mas acredite: esta nova classe de comando irá retornar a configuração de conexão;

F. No passo 7 o invocador chama o método Execute da classe TBuscarConfiguracoesCommand, que por usa vez irá dar um jeito de buscar as configurações existentes. Você pode ver isto no passo 8;

G. Quando a classe TBuscarConfiguracoesCommand encontra as configurações existentes, o método Execute chamado no passo 7 termina e o invocador registra que o último objeto de comando executado foi ComandoBuscar (passo 9). Fazendo este registro, o invocador consegue saber qual comando desfazer quando isto for solicitado;

H. Agora a classe TSalvarConfiguracoesCommand já tem uma cópia da configuração atual: no passo 10 ela preserva esta cópia para que seja possível voltar estes valores quando o invocador chamar o método Undo. Na sequência o processo de salvar a nova configuração finalmente é executado, veja o passo 11;

I. Quando TSalvarConfiguracoesCommand termina de salvar a nova configuração, o método Execute que havia sido invocado no passo 4 termina e o invocador registra o úlitmo comando invocado: ComandoSalvar (passo 12). Lembre-se que o invocador já havia registrado um outro comando – Veja o item G desta lista. Isto sifnifica que agora o invocador possui uma lista com dois comandos que podem ser desfeitos;

J. Passo 13 - após passar algum tempo o usuário deseja desfazer as alterações que ele fez nas configurações de conexão. Para isso ele irá clicar em um botão "Desfazer". O evento OnClick deste botão irá fazer uma chamada ao método Undo (passo 14), que também foi criado na classe TCommandInvoker. Eu ainda não mostrei como é este novo método: falta pouco para você ver como ele funciona;

K. No passo 15 o método Undo da classe TCommandInvoker procura pelo último comando executado e no passo 16 chama o método Undo do comando encontrado. Lembre-se que o registro dos comandos executados é feito após a chamada do método Execute de cada comando – leia o item I desta lista novamente;

L. No passo 17 o método Undo da classe TSalvarConfiguraçõesCommand é executado: internamente este método apenas salva a configuração que ele havia buscado no item E desta lista;

M. Por fim, o invocador remove o comando que foi desfeito da lista de comandos executados – passo 18.

A lógica para desfazer um comando apresentada aqui não passa de uma simples ideia, isto significa que talvez você use um algoritmo diferente do que foi apresentado no diagrama da **Figura 2**. Se isto acontecer, não significa que você está errado.

Ações reversíveis – uma ideia implementada

Este é o momento em que a ideia apresentada no tópico anterior começa a ser implementada, ou seja, a ideia começa a virar código. No item E da explanação anterior você viu que será necessário criar mais uma classe de comando:

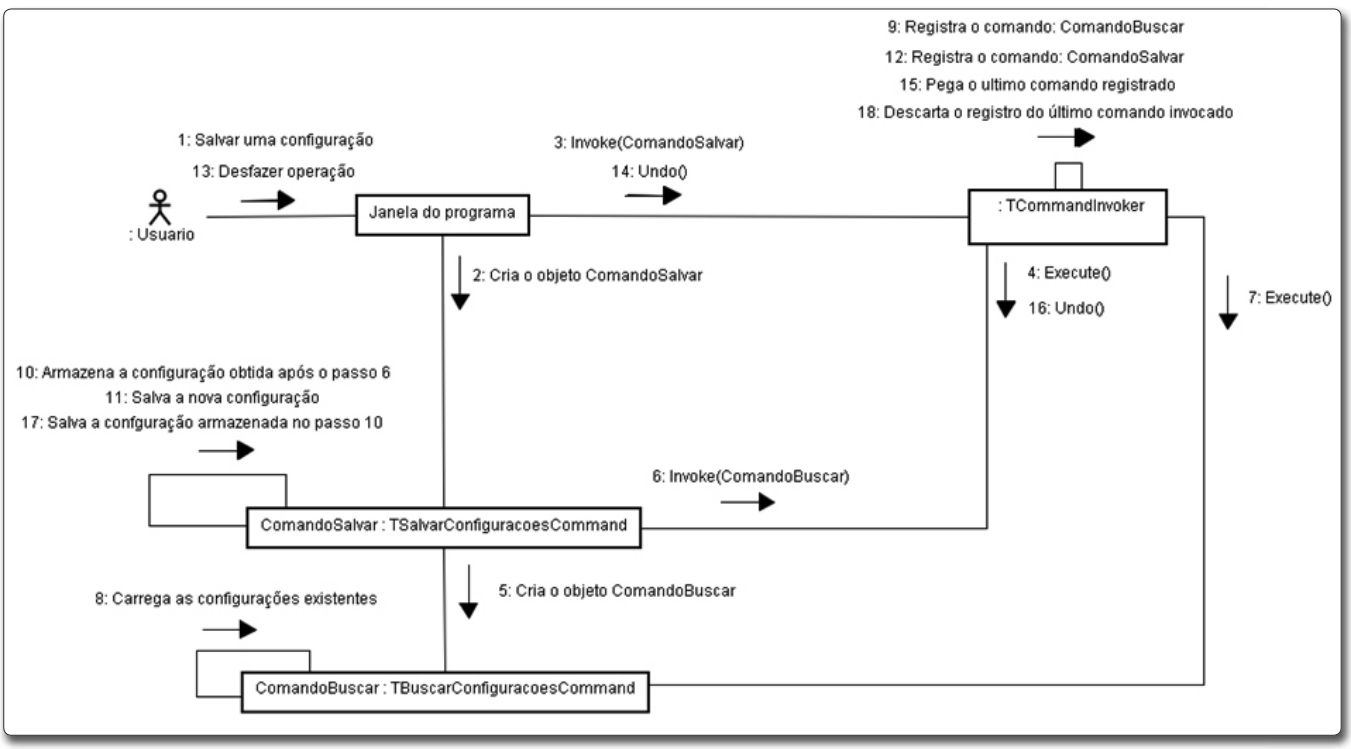


Figura 2. Executando e desfazendo uma operação

TBuscarConfiguracoesCommand. Esta classe de comando será a responsável por retornar a configuração de conexão atual. Veja como ela faz isto na **Listagem 15**.

Listagem 15. Comando para buscar a configuração atual

```
1  unit BuscarConfiguracoesCommand;
2  Interface
3  uses Command, Configurações;
4  Type
5    TBuscarConfiguracoesCommand = class(TInterfacedObject,
     ICommand)
6    Private
7      ConfiguracoesCarregadas: ^TConfiguracoes;
8    Public
9      constructor Create(out Configuracoes: TConfiguracoes);
10     procedure Execute;
11     procedure Undo;
12   end;
13 Implementation
14 uses Registry, Windows, SysUtils, CommandInvoker;
15 { TBuscarConfiguracoesCommand }
16 constructor TBuscarConfiguracoesCommand.Create(
17   out Configuracoes: TConfiguracoes);
18 Begin
19   inherited Create;
20   Configuracoes := TConfiguracoes.Create;
21   ConfiguracoesCarregadas := @Configuracoes;
22 end;
23 Procedure TBuscarConfiguracoesCommand.Execute;
24 Var
25   Registro: TRegistry;
26 Begin
27   {**
28    * Não seria necessário copiar as próximas três linhas da
        classe
29    * TSalvarConfiguracoesCommand, se também fosse usado o
        padrão Factory.
30    *}
31   Registro := TRegistry.Create;
32   Registro.RootKey := HKEY_LOCAL_MACHINE;
33   Registro.OpenKey('Software\Padrão Command', True);
34   ConfiguracoesCarregadas.SetCaminhoDeConexao(
       Registro.ReadString('Caminho'));
35   ConfiguracoesCarregadas.SetUsuario(Registro.
       ReadString('Usuario'));
36   ConfiguracoesCarregadas.SetSenha(Registro.
       ReadString('Senha'));
37   Registro.CloseKey;
38   FreeAndNil(Registro);
39 end;
40 Procedure TBuscarConfiguracoesCommand.Undo;
41 Begin
42   TCommandInvoker.Undo;
43 end;
44 end.
```

Vamos ao que tem de importante na **Listagem 15**. Na linha 5 você tem a confirmação de que esta classe é uma classe de comando, pois ela implementa a interface ICommand. Entre as linhas 16 e 22 existe a implementação do método construtor. Esse método recebe um parâmetro especificado como out. Quando você vê um parâmetro com especificação out, significa que o método que você vai executar irá colocar alguma informação neste parâmetro para você usar após a execução do método. Isto é mais comum em métodos da API do Windows. Mas, por que esta classe deseja ter um parâmetro out no seu construtor? Para entender isto você precisa lembrar que todo o resto do sistema, exceto a classe TCommandInvoker, não conhece nada sobre as classes de comando. Tudo que o resto do programa pode fazer é usar o método construtor para criar objetos destas classes. Então pense um pouco: se você tem uma classe, cujo único método que você pode executar é o próprio método construtor, como você faz para esta classe retornar o resultado do que ela processou? Como você faz para que a classe de comando responsável por buscar as configurações retorne a configuração encontrada? Simples: você usa um parâmetro out.

Na linha 21 a classe TBuscarConfiguracoesCommand guarda o ponteiro para a variável do parâmetro out onde a configuração encontrada será colocada. A implementação do método Execute está entre as linhas 23 e 39. Olhe com atenção nas linhas 34, 35 e 36. Veja que os valores da configuração localizada são colocados na variável para onde o ponteiro está apontando. Neste caso está apontando para a variável do tipo out recebido no constructor da linha 16. A implementação do método Undo está um pouco abaixo, entre as linhas 40 e 43. Responda rápido, como você desfaz um procedimento de busca? Resposta: Você não desfaz. Por isto o método Undo da classe TBuscarConfiguracoesCommand não faz nada além de pedir para que a ação anterior seja desfeita.

Com a classe responsável por buscar uma configuração em mãos é preciso passar para o próximo passo, alterar o método Execute da classe TSalvarConfiguracoesCommand. Como foi explicado no item E anteriormente, antes de salvar uma nova configuração é preciso buscar os valores da configuração atual para que estes valores sejam restaurados quando o método Undo for chamado. Veja como fica o método Execute na **Listagem 16**.

Listagem 16. Alterando o método Execute

```
1  procedure TSalvarConfiguracoesCommand.Execute;
2  Var
3    Registro: TRegistry;
4  Begin
5    {**
6     * Antes de alterar a configuração, é armazenado em uma
        variável o
7     * valor da configuração atual. Assim o método Undo pode ser
8     * implementado com mais facilidade.
9     *}
10    TCommandInvoker.Invoke(
11     TBuscarConfiguracoesCommand.Create(UltimaConfiguracao));
12    {**
13     * As próximas três linhas poderiam ser apenas uma,
14     * se também fosse usado o padrão Factory.
15     *}
16    Registro := TRegistry.Create;
17    Registro.RootKey := HKEY_LOCAL_MACHINE;
18    Registro.OpenKey('Software\Padrão Command', True);
19    Registro.WriteString('Caminho', ConfiguracoesParaSalvar.
       GetCaminhoDeConexao);
20    Registro.WriteString('Usuario', ConfiguracoesParaSalvar.
       GetUsuario);
21    Registro.WriteString('Senha', ConfiguracoesParaSalvar.
       GetSenha);
22    Registro.CloseKey;
23    FreeAndNil(Registro);
24 end;
```

A única mudança está na linha 10 e 11. Nestas linhas o comando TBuscarConfiguracoesCommand é invocado e a configuração que ele encontrar será usada para implementar o método Undo.

Observe na linha 18 da **Listagem 14** que o método Undo foi declarado, mas não foi implementado, ou seja, ele não faz nada. Agora já é possível implementá-lo, como mostra a **Listagem 17**.

Como você provavelmente deve ter imaginado, o método Undo é bem simples. Ele simplesmente salva os valores da configuração que o método Execute havia recuperado. Você deve ter notado também que os métodos Undo e Execute são muito parecidos e existe uma grande quantidade de código duplicado entre estes

métodos. Quando digo código duplicado não estou querendo dizer que as linhas de código são idênticas. Pois não são. Veja que o nome da variável usada na linha 19 das **Listagens 16** e **17** é diferente. Quando digo código duplicado estou me referindo ao objetivo do código, ou seja, existe uma parte do código nas **Listagens 16** e **17** que possui o mesmo objetivo: salvar uma configuração. Para resolver este problema de duplicação vou criar outro método dentro desta classe cuja responsabilidade dele será salvar uma configuração. Veja como ficam os métodos Undo e Execute após a criação deste terceiro método na **Listagem 18**.

Nota

Para economizar espaço, alguns detalhes não aparecem na **Listagem 19**. São eles: a variável UltimaConfiguracao usada na linha 4 é do tipo TConfiguracoes e está declarada na sessão private da classe. As units CommandInvoker e BuscarConfiguracoesCommand foram adicionadas na uses da unit SalvarConfiguracoesCommand.pas.

Listagem 17. Método Undo da classe TSalvarConfiguracoesCommand

```
1  procedure TSalvarConfiguracoesCommand.Undo;
2  Var
3     Registro: TRegistry;
4  Begin
5     {**
6     * Salva os valores da última configuração se existir uma referência
7     * válida para estes valores. Caso contrário não faz nada,
          porque significa
8     * que o método Undo já foi executado.
9     *}
10     if Assigned(UltimaConfiguracao) then
11    Begin
12       {**
13        * As próximas três linhas poderiam ser apenas uma,
14        * se também fosse usado o padrão Factory.
15        *}
16       Registro := TRegistry.Create;
17       Registro.RootKey := HKEY_LOCAL_MACHINE;
18       Registro.OpenKey('Software\Padrão Command', True);
19       Registro.WriteString('Caminho', UltimaConfiguracao.
          GetCaminhoDeConexao);
20       Registro.WriteString('Usuario', UltimaConfiguracao.
           GetUsuario);
21       Registro.WriteString('Senha', UltimaConfiguracao.
           GetSenha);
22       Registro.CloseKey;
23       FreeAndNil(Registro);
24       FreeAndNil(UltimaConfiguracao);
25    end;
26 End;
```

Listagem 18. Refatorando o código: extrair método

```
1  procedure TSalvarConfiguracoesCommand.Execute;
2  Begin
3     TCommandInvoker.Invoke(
4       TBuscarConfiguracoesCommand.Create(UltimaConfiguracao));
5     Salvar(ConfiguracoesParaSalvar);
6    end;
7    procedure TSalvarConfiguracoesCommand.Undo;
8    Begin
9     if Assigned(UltimaConfiguracao) then
10    Begin
11      Salvar(UltimaConfiguracao);
12      FreeAndNil(UltimaConfiguracao);
13    end;
14  end;
15  procedure TSalvarConfiguracoesCommand.Salvar(Configuracao:
      TConfiguracoes);
16  Var
17    Registro: TRegistry;
18  Begin
19   Registro := TRegistry.Create;
20   Registro.RootKey := HKEY_LOCAL_MACHINE;
21   Registro.OpenKey('Software\Padrão Command', True);
22   Registro.WriteString('Caminho',Configuracao.
       GetCaminhoDeConexao);
23   Registro.WriteString('Usuario', Configuracao.GetUsuario);
24   Registro.WriteString('Senha', Configuracao.GetSenha);
25   Registro.CloseKey;
26   FreeAndNil(Registro);
27 end;
```

A **Listagem 18** apresenta um código bem mais simples, você concorda? O código que estava duplicado agora está dentro de um único método (na linha 15) que pode salvar qualquer configuração. Nas linhas 5 e 11 é possível ver o método Salvar sendo usado. Um detalhe que não aparece na **Listagem 18** é que o método Salvar foi declarado dentro da sessão private da classe, uma vez que ele só tem utilidade para a própria classe TSalvarConfiguracoesCommand. Não faria sentido se este método tivesse um nível de visibilidade mais alto.

Se você tentar compilar o projeto de exemplo agora, não irá conseguir. Está faltando encapsular o método Undo dos comandos assim como foi feito para o comando Execute. Veja como fica a classe TCommandInvoker após a implementação do método Undo que serve para encapsular a chamada do método dos comandos (**Listagem 19**).

Listagem 19. Classe invocadora também encapsula o método Undo

```
1  unit CommandInvoker;
2  Interface
3  uses Command, Contnrs;
4  Type
5    TCommandInvoker = class
6   Private
7     class function GetPilhaDeComandosExecutados: TStack;
8   Public
9     class procedure Invoke(Comando: ICommand);
10     class procedure Undo;
11   end;
12 Implementation
13 uses ComServ;
14 {**
15  * Esta variável poderia estar na sessão private desta
      classe, se o
16  * fosse aplicado o padrão Singleton
17  *}
18 Var
19   PilhaDeComandosExecutados: TStack;
20  { TCommandInvoker }
21 class function TCommandInvoker.GetPilhaDeComandosExecutados:
     TStack;
22 Begin
23   {**
24    * Este método chamado de Factory Method: responsabilidade dele
25    * é criar e devolver alguma coisa, que neste caso é um objeto
26     * da classe TStack.
27     *}
28   if not Assigned(PilhaDeComandosExecutados) then
29     PilhaDeComandosExecutados := TStack.Create;
30   Result := PilhaDeComandosExecutados;
31 end;
32 class procedure TCommandInvoker.Invoke(Comando: ICommand);
33 Begin
34   Comando.Execute;
35   Comando._AddRef;
36   GetPilhaDeComandosExecutados.Push(Pointer(Comando));
37 end;
38 class procedure TCommandInvoker.Undo;
39 Var
40   UltimoComando: ICommand;
41  Begin
42   //Se existir algum comando para desfazer
43   if GetPilhaDeComandosExecutados.Count > 0 then
44   Begin
45     UltimoComando := ICommand(GetPilhaDeComandosExecutados.Pop);
46     UltimoComando.Undo;
47     UltimoComando._Release;
48   end;
49  end;
50  end.
```

O novo código da classe CommandInvoker apresentando na **Listagem 19** possui uma lógica realmente muito simples. Em primeiro lugar foi declarada uma variável do tipo TStack na linha 19. Objetos TStack gerenciam uma lista como se fosse um TStringList. A diferença é que TStringList gerencia uma lista de strings enquanto que TStack gerencia uma

lista de ponteiros para objetos. A cada chamada do método Invoke, a lista de ponteiros recebe um ponteiro para o comando que foi invocado (linha 36 da **Listagem 19**). Assim quando o método Undo for executado ele verifica se a lista possui algum ponteiro (linha 43) e caso exista, é feita a chamada do método Undo do comando para o qual o ponteiro está apontando.

Nota

Para que o garbage collector não destrua os comando automaticamente após eles serem executados é necessário chamar o método _AddRef de cada comando – veja linha 35 da **Listagem 19.** Isto permite que você controle o momento em que o comando pode ou não ser destruído. Após chamar _AddRef você precisa chamar o método _Release (linha 47 da **Listagem 19**) para destruir o objeto, caso ninguém mais esteja usando ele em outro lugar.

Para finalizar, é hora de colocar tudo isto em teste. Volte ao arquivo DPR (Project>View Source) e digite o código da **Listagem 20**.

Listagem 20. Último teste

```
1  {$R *.res}
2  Var
3    Config: TConfiguracoes;
4    ConfigRecuperada: TConfiguracoes;
5  Begin
6    Application.Initialize;
7    Config := TConfiguracoes.Create;
8    Config.SetCaminhoDeConexao('LOCALHOST:C:\Base.fdb');
9    Config.SetUsuario('SYSDBA');
10   Config.SetSenha('masterkey');
11   TCommandInvoker.Invoke(TSalvarConfiguracoesCommand.
       Create(Config));
12   TCommandInvoker.Invoke(TBuscarConfiguracoesCommand.
       Create(ConfigRecuperada));
13   Application.MessageBox(PChar('Configuração atual: ' +
14     ConfigRecuperada.GetCaminhoDeConexao), 'Aviso', 48);
15   Config.SetCaminhoDeConexao('SERVER:Banco_Backup');
16   TCommandInvoker.Invoke(TSalvarConfiguracoesCommand.
       Create(Config));
17   TCommandInvoker.Invoke(TBuscarConfiguracoesCommand.
       Create(ConfigRecuperada));
18   Application.MessageBox(PChar('Nova configuração está
     apontando para: '+
19     ConfigRecuperada.GetCaminhoDeConexao), 'Aviso', 48);
20   Application.MessageBox(
21     'Clique em Ok para desfazer a alteração de
         configuração','Aviso', 48);
22   TCommandInvoker.Undo;
23   ConfigRecuperada := nil;
24   TCommandInvoker.Invoke(TBuscarConfiguracoesCommand.
       Create(ConfigRecuperada));
25   Application.MessageBox(PChar('A configuração foi desfeita. ' +
     ' O valor atual ' +
26   'voltou a ser: ' + ConfigRecuperada.GetCaminhoDeConexao),
     'Aviso', 48);
27   Application.Run;
28 end.
```

Entenda o que foi feito na **Listagem 20**. Entre as linhas 7 e 11 é feito a definição da configuração cujo caminho de conexão é LOCALHOST:C:\Base.fdb. Na linha 12 usei o comando TBuscarConfiguracoesCommand para verificar se a configuração definida entre as linhas 7 e 11 realmente foi salva: na linha 13 e 14 uma mensagem é exibida para mostrar o caminho da conexão encontrada pelo comando de busca. Em seguida, na linha 15 e 16 o caminho da conexão foi alterado para SERVER:Banco_Backup. Para garantir que a configuração realmente foi alterada, o comando de busca é executado novamente na linha 17 e nas linhas 18 e 19, é apresentada uma mensagem dizendo qual foi o caminho da conexão encontrada. Por último o comando Undo da classe TCommandInvoker é chamado (linha 22) e para garantir que a configuração voltou a ser o que era antes da alteração feita na linha 17, o comando de busca é executado pela última vez na linha 24 e na linha 25 o caminho da conexão encontrada é exibido novamente. Neste caso o caminho da conexão deve voltar a ser LOCALHOST:C:\Base.fdb definido na linha 7 e 11. Execute o seu exemplo e veja o resultado.

Conclusão

Você aprendeu neste artigo o poder que existe no encapsulamento e viu o Padrão Command utilizando este poder na chamada de métodos. Este padrão permite que você use algumas outras características que não foram apresentados neste artigo, como por exemplo, a criação de macrocomandos que possibilitam a uma única classe de comando (denominado macrocomando), invocar vários outros comandos. Além disso, o Padrão Command permite o enfileiramento de solicitações. Com esta técnica o processamento de cada comando pode ser executado muito tempo depois que a classe cliente criou o objeto de comando. Esta é uma característica que pode ser útil em mais de uma situação, das quais posso citar aquela onde você precise criar temporizadores, pools de processamento ou fila de tarefas.

No projeto de exemplo disponível pra download, além das classes apresentadas no artigo, você vai encontrar outra classe para excluir a configuração atual e para simular um sistema mais real. Também existe um formulário de onde os comandos são executados. Não deixe de baixá-lo para conferir. ●

Links

Diagrama de colaboração

http://pt.wikipedia.org/wiki/Diagrama_de_colaboração

Refatoração

http://pt.wikipedia.org/wiki/Refatoração

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

Generics, RTTI e Custom Attributes

Desenvolva um mini framework de validação

Do que trata o artigo

Explorar o uso dos recursos aprimorados no Delphi 2010, Generics, RTTI e Custom Attributes. Essa exploração se dá na explicação conceitual de cada item, com exemplos inicialmente simples e práticos para posteriormente unir esses recursos e desenvolver algo mais complexo, o mini framework de validação que poderá ser utilizado e reutilizado em vários projetos.

Para que serve

Ao compreender o uso de Generics, RTTI e Custom Attributes, será possível utilizá-los para o desenvolvimento de frameworks que venham agilizar o dia-a-dia. Frameworks que poderão ser reutilizados entre projetos distintos ou mesmo que não se deseje criar um framework, será possível centralizar rotinas básicas como validação, acesso a dados etc.

Em que situação o tema é útil

Para mostrar a utilidade, desenvolvemos um mini framework de validação, contudo seu uso pode ser muito amplo. Com Generics é possível especializar listas como também criar ordenadores de listas. Com RTTI podemos explorar um projeto em tempo de execução, podendo até mesmo executar um método por seu nome, gerando certa flexibilidade ao código. Podemos também adicionar informações em campos, classes, propriedades através dos Custom Attributes.

Paulo Quicoli
pauloquicoli@gmail.com
Editor técnico das revistas ClubeDelphi, .Net Magazine e WebMobile Magazine, formado em processamento de dados pela FATEC-TQ. Atua como analista e desenvolvedor da ControlM Informática (www.controlm.com.br). Utiliza Delphi desde de sua primeira versão, entusiasta do desenvolvimento orientado a objetos, tem publicado vários artigos sobre o assunto. Blog: www.pauloquicoli.spaces.live.com

Resumo do DevMan

Vamos mostrar na prática o uso de três recursos melhorados no Delphi 2010: Generics, RTTI e Custom Attributes. Vamos entender como cada um funciona particularmente e então unir o poder de cada um para o desenvolvimento de um framework de validação. Com esse framework será possível validar se um TEdit pode ser vazio. Se uma quantidade mínima de caracteres foi informada e muito mais, além de poder mostrar isso ao usuário de uma forma elegante.

Vamos construir um pequeno framework de validação que poderá ser utilizado em suas classes e até mesmo nos componentes inseridos em um determinado formulário. Para isso vamos fazer uso dos recursos novos do Delphi 2010: Generics, RTTI e Custom Attributes. Portanto vamos compreender o que são eles.

Generics

A introdução do Generics na linguagem Delphi permite ao desenvolvedor a criação de parâmetros de tipo. Mas o que é isso exatamente? Normalmente quando definimos uma função ou procedimento que receba parâmetros esses são passados como instâncias. Observe o seguinte código:

```
procedure EscreverNaTela(const Texto: TTextoDeAjuda);
```

O procedimento EscreverNaTela recebe como parâmetro um objeto do tipo TTextoDeAjuda, ou seja, recebe uma instância dessa classe. Nesse caso é possível passar para o método apenas objetos do tipo esperado ou seus descendentes. Ao utilizarmos Generics é possível passar um tipo como parâmetro, e esse tipo será resolvido em tempo de execução, permitindo que o procedimento seja parametrizado. O mesmo procedimento usando Generics teria a seguinte assinatura:

```
procedure EscreverNaTela<T>;
```

Onde T é o tipo de dado. O Delphi já traz em sua estrutura de classes algumas prontas para utilizar Generics. Vamos observar a classe TList<T>, **Listagem 1**.

Listagem 1. Estrutura da classes TList<T>

```
type
TList<T> = class(TEnumerable<T>)
// ...
public
// ...
function Add(const Value: T): Integer;
procedure Insert(Index: Integer; const Value: T);
function Remove(const Value: T): Integer;
procedure Delete(Index: Integer);
procedure DeleteRange(AIndex, ACount: Integer);
function Extract(const Value: T): T;
procedure Clear;
property Count: Integer read FCount write SetCount;
property Items[Index: Integer]: T read GetItem write SetItem;
default;
end;
```

Observe que os métodos foram adaptados para se trabalhar com tipo genérico, permitindo que um TList seja tipado. Antes disso era necessário realizar operações de boxing e unboxing (falaremos mais sobre isso a seguir) para armazenar um tipo específico, o que poderia levar a erros. Para exemplificar isso, vamos construir nosso primeiro exemplo.

Utilizando Generics

Vamos ter uma aplicação VCL Forms, e nela dois botões. Nesse primeiro botão vamos mostrar o uso de um TList comum e no segundo o uso do TList<T>. Crie então a aplicação, insira os dois botões. Logo abaixo da declaração da classe do formulário vamos criar uma classe TPessoa, como mostra **Listagem 2**.

Listagem 2. Classe TPessoa

```
TPessoa = class
  public
    Nome: string;
    constructor Create(NomePessoa: string); reintroduce;
    function ToString: string; override;
end;
```

Veja que é uma classe simples. Temos um construtor, uma propriedade e sobrecarregamos o método ToString, que existe originalmente em TObject. Esse método ToString pode ser utilizado para retornar o objeto no formato string, conforme vemos na implementação da classe na **Listagem 3**.

Listagem 3. Implementando a classe TPessoa

```
function TPessoa.ToString: string;
begin
  result := Nome;
end;

constructor TPessoa.Create(NomePessoa: string);
begin
  inherited Create;
  Nome := NomePessoa;
end;
```

Mais uma vez o código é bem simples. No construtor apenas instanciamos o objeto, ao chamar o Create da classe base TObject e em seguida passamos para o campo Nome o valor do parâmetro NomePessoa. Já no método ToString apenas retornamos o valor de Nome.

Vamos agora ao primeiro botão, criar uma lista e nela adicionar três objetos TPessoa e posteriormente exibir em um diálogo o conteúdo da lista. Essa operação pode ser vista na **Listagem 4**.

Listagem 4. Utilizando um TList para conter objetos

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Lista: TList;
  Texto: string;
  I: Integer;
begin
  Lista := TList.Create;
  Lista.Add(TPessoa.Create('Paulo'));
  Lista.Add(TPessoa.Create('José'));
  Lista.Add(TPessoa.Create('Maria'));

  for I := 0 to Lista.Count - 1 do
  Texto := Texto + TPessoa(Lista[I]).Nome + #13;

  ShowMessage(Texto);
end;
```

Criamos a variável Lista do tipo TList e a instanciamos. Depois, utilizamos o método Add para inserir objetos do tipo TPessoa. Para listar seus nomes, criamos uma variável Texto do tipo string e em um comando "for" percorremos a lista. Observe que somos obrigados a fazer uma conversão de tipo para que seja possível acessar o campo Nome do objeto que está dentro da lista. Estamos convertendo o que está dentro da lista para TPessoa. Mas aí você pode se perguntar: Se eu inseri na lista um TPessoa, porque que para acessá-lo preciso converter para TPessoa novamente?

Esse é famoso boxing e unboxing. O método Add de TList espera como parâmetro um Pointer, ponteiro, e não um objeto do tipo TPessoa. Por isso, ao inserir é feito um boxing.

O objeto é transformado para o tipo ponteiro, e posteriormente quando vamos acessá-lo realizamos o inverso, o unboxing. Transformamos de ponteiro para TPessoa. Essas operações de transformações de tipo podem causar alguma perda de desempenho e, o mais agravante em minha opinião, é sujeita à falha humana.

Como Add recebe um ponteiro como parâmetro, é possível passar qualquer objeto, o que pode levar a erros quando formos acessar os elementos da lista. Para mostrar o erro, apenas adicione logo abaixo do último Add o seguinte código:

```
Lista.Add(self);
```

Execute agora o programa e você verá o erro aparecendo. Isso porque Self, que no contexto atual representa o TForm1, não herda de TPessoa, e no momento que vamos acessá-lo em Lista tentamos convertê-lo para esse tipo.

Vamos agora utilizar Generics. Adicione o código da **Listagem 5** no evento OnClick do segundo botão.

Listagem 5. Utilizando Generics

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Lista : TList<TPessoa>;
  Texto: string;
  I: Integer;
begin
  Lista := TList<TPessoa>.Create;
  Lista.Add(TPessoa.Create('Paulo'));
  Lista.Add(TPessoa.Create('José'));
  Lista.Add(TPessoa.Create('Maria'));

  for I := 0 to Lista.Count - 1 do
   Texto := Texto + Lista[I].Nome + #13;

  ShowMessage(Texto);
end;
```

Nota

Para utilizar o List<T> é necessário incluir na cláusula uses a unit Generics.Collections

A primeira mudança nítida é que a variável Lista não é mais TList e sim TList<TPessoa>. Veja que ao obter os objetos da Lista (no "for"), não é mais necessário realizar a conversão de tipos. Isso porque Lista sabe agora que contém elementos do tipo TPessoa e não mais meros ponteiros, ou seja, não temos mais operações de boxing e unboxing.

Ao definirmos dessa maneira, estamos tipando o TList de tal forma que ele aceitará apenas objetos do tipo TPessoa em seus métodos. Tente por exemplo adicionar um novo elemento à Lista da seguinte forma:

```
Lista.Add(self);
```

Tente compilar. O Delphi não deixará compilar. Com isso evitamos falhas como a mostrada anteriormente. Nós criamos uma lista de um tipo específico nosso, de uma classe nossa, mas é possível utilizá-lo para armazenar strings: TList<String>, inteiros: TList<integer> e outros tipos.

RTTI

A capacidade que um ambiente de desenvolvimento tem em se auto conhecer, podendo recuperar os métodos, campos e propriedades de suas classes é chamada de reflexão. Esse recurso foi aprimorado no Delphi 2010. Antes para que uma informação pudesse estar disponível para reflexão, ela deveria estar na seção published da classe, essa imposição não existe mais. Podemos obter informações de métodos, propriedades que estejam na seção public e campos em qualquer seção. Para entendermos melhor, vamos criar uma aplicação bem simples. Inicie um novo projeto do tipo VCL Forms Application. Deixe o formulário principal como na **Figura 1**.

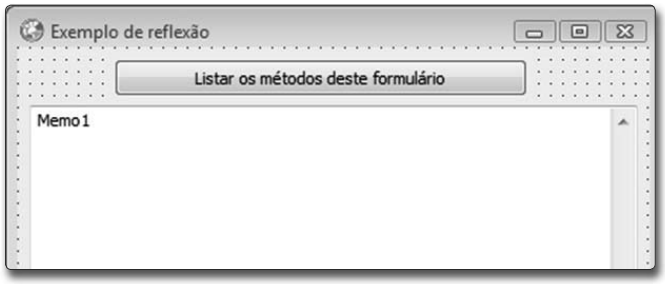


Figura 1. Formulário principal

Como mostra a **Figura 1**, vamos listar no Memo1 os métodos existentes no formulário. Dê um duplo clique no botão e vamos implementar seu evento OnClick como mostra a **Listagem 6**.

Listagem 6. Extraindo os métodos

```
procedure TForm1.Button1Click(Sender: TObject);
var
  contexto : TRttiContext;
  metodo : TRttiMethod;
  tipo: TRttiType;
begin
  Memo1.Clear;
  contexto := TRttiContext.Create;
  try
    tipo := contexto.GetType(self.ClassInfo);
    for metodo in tipo.GetMethods do
        Memo1.Lines.Add('Método: ' + metodo.Name);

  finally
    contexto.Free;
  end;
end;
```

Nota

Adicione na seção Uses a unit Rtti.pas

Todo o trabalho de reflexão foi organizado em classes para facilitar seu uso. Essas classes estão na unit Rtti. Seu código fonte é disponibilizado e você pode abrir para dar uma olhada no código. Posteriormente criamos algumas variáveis de alguns tipos constantes na Rtti.pas. A primeira, chamada contexto, é do tipo TRttiContext. Ela representa um contexto de reflexão, controlando em si os objetos que são criados. Isso significa que, ao liberarmos o contexto da memória, todos objetos de reflexão utilizados no contexto também são liberados. Por isso que seu uso é feito em um bloco try finally.

Na sequência utilizamos o método GetType de contexto, passando para ele o Self que nesse momento é o formulário. GetType nos retorna um objeto do tipo TRttiType que representa o tipo obtido (TForm1). Essa classe disponibiliza alguns métodos que nos permite explorar a classe TForm1, veja **Listagem 7**.

Listagem 7. Métodos de reflexão

```
function GetMethods: TArray<TRttiMethod>; overload; virtual;
function GetFields: TArray<TRttiField>; virtual;
function GetProperties: TArray<TRttiProperty>; virtual;
function GetMethod(const AName: string): TRttiMethod; virtual;
function GetMethods(const AName: string): TArray<TRttiMethod>;
overload; virtual;
function GetField(const AName: string): TRttiField; virtual;
function GetProperty(const AName: string): TRttiProperty;
virtual;
```

Como pode ser visto, é possível extrair os métodos, campos e propriedades. Continuando utilizamos o método GetMethods que retorna um array de TRttiMethod. Essa classe representa os métodos encontrados no tipo passado como parâmetro e assim, podemos acessar algumas propriedades desses métodos. Isso é visto no loop "for in" que aplicamos. Temos uma variável do tipo TRttiMethod e a utilizamos para percorrer a lista retornada, adicionando no Memo1 o nome do método atual. Execute a aplicação e veja que até mesmo o método do evento Click do botão é listado (**Figura 2**)

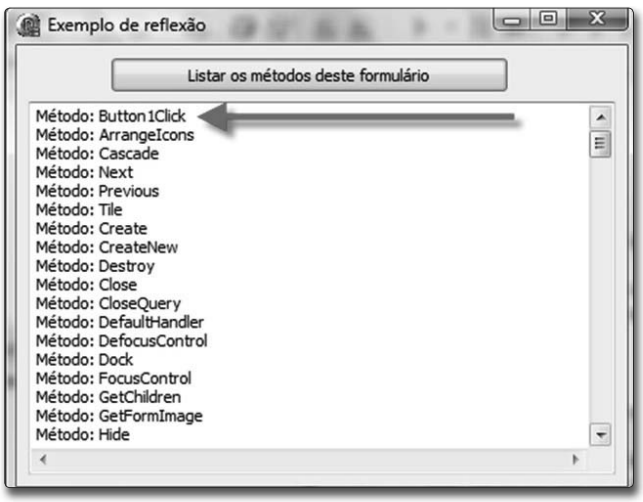


Figura 2. Métodos listados

Custom Attributes

Eles são uma forma de se acrescentar informações adicionais a uma propriedade, método, classe, campo ou até mesmo variáveis. E essas informações também podem ser obtidas pela RTTI. Um custom attribute nada mais é que uma classe herdada de TCustomAttribute com propriedades específicas que podem ser utilizadas. Vamos criar um exemplo que mostre seu uso. Crie uma nova aplicação do tipo VCL Forms e deixe o formulário principal como mostra a **Figura 3**.

Agora vamos criar a seguinte classe (**Listagem 8**) antes da declaração da classe TForm1. Essa classe é o nosso Custom Attribute.

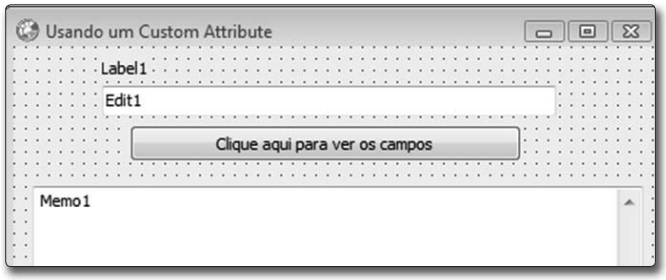


Figura 3. Testando atributos

Listagem 8. Custom Attribute

```
THintAttribute = class(TCustomAttribute)
private
  FHint: string;
  procedure SetHint(const Value: string);
published
  public
    constructor Create(Hint: string);
    property Hint: string read FHint write SetHint;
end;
```

Existe uma padronização nos nomes de atributos customizáveis. Como pode ser visto, temos o nome da classe + o sufixo Attribute, essa é uma convenção, uma boa prática a ser seguida quando se desenvolve custom attributes. Observe que é uma classe comum, herdando de TCustomAttribute. Temos um construtor que recebe como parâmetro uma string que é posteriormente passada para o campo FHint. Vamos usar este atributo para adicionar um texto explicativo aos controles que estão inseridos no formulário. Com a classe pronta, vamos apenas decorar os campos, como mostra a **Listagem 9**.

Listagem 9. Aplicando o atributo

```
TForm1 = class(TForm)
  [THintAttribute('Eu sou o Edit 1')]
  Edit1: TEdit;

  [THintAttribute('Eu sou o Label que diz o que o Edit 1 é')]
  Label1: TLabel;

  [THintAttribute('Sou um botão, se clicar em mim listo os
   campos')]
  Button1: TButton;

  [THintAttribute('Em mim você ve os custom attributes')]
  Memo1: TMemo;

private
  { Private declarations }
public
  { Public declarations }
end;
```

Veja que não tem segredo em como aplicar o atributo. O colocamos entre colchetes sobre o elemento no qual desejamos adicionar mais informações. Agora falta apenas extraí-los por RTTI. Dê um clique duplo no botão e no evento OnClick adicione o seguinte código, **Listagem 10**.

De forma muito similar ao exemplo anterior sobre RTTI, temos o TRttiContext e TRttiType. Porém temos dois novos personagens: TRttiField e TCustomAttribute. O primeiro representa um campo existente na classe a ser inspecionada, o segundo um possível atributo que o campo possa possuir. Então criamos o contexto de reflexão, passando o Self para obtermos seu tipo. Com isso podemos extrair os campos do tipo através do método GetFields para então, percorrer cada

TRttiField e extrair dele seus atributos pelo método GetAttributes. Se o campo possuir um custom attribute sobre ele, este será retornado e então testamos se este é do tipo THintAttribute. Se for, adicionamos no Memo1 o nome do campo mais a informação adicional do atributo. Rode o exemplo, o resultado deve ser como mostra a **Figura 4**.

Listagem 10. Recuperando os atributos

```
procedure TForm1.Button1Click(Sender: TObject);
var
  contexto : TRttiContext;
  campo : TRttiField;
  tipo: TRttiType;
  atributo: TCustomAttribute;
begin
  Memo1.Clear;
  contexto := TRttiContext.Create;
  try
    tipo := contexto.GetType(self.ClassInfo);
    for campo in tipo.GetFields do
       for atributo in campo.GetAttributes do
         if (atributo is THintAttribute) then
          Memo1.Lines.Add('Campo: ' + campo.Name + ' Informação:
          '+ (atributo as THintAttribute).Hint);

  finally
    contexto.Free;
  end;
end;
```

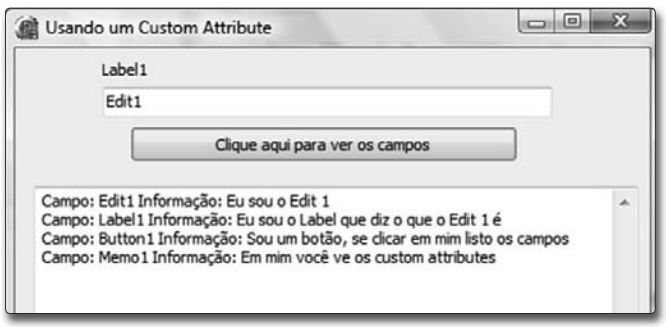


Figura 4. Extraindo os atributos

Misturando tudo

Vamos agora misturar todos esses recursos e construir um pequeno framework de validação inspirado no Validation Application Block da Microsoft. A ideia é poder aplicar, usando custom attributes, regras simples de validação em controles VCL e até mesmo em nossas próprias classes de negócio para então obtermos uma lista com os erros. Vamos montar a seguinte arquitetura de classes (**Figura 5**).

Como podemos ver, temos uma classe chamada ValidationAttribute, ela herda de TCustomAttribute. Esta será a base para criar nossos validadores. Ela expõe uma propriedade TextMessage do tipo string que armazenará o texto informativo do erro. Também temos um método abstrato IsValid. Este método será implementado pelos validadores e conterá a lógica de cada tipo de validação.

Crie um novo projeto do tipo VCL Forms Application e adicione uma nova unit de nome Validation. Nela adicione o código da **Listagem 11** referente à classe ValidationAttribute e sua implementação.

Como vemos, nada difícil ainda. Apenas um construtor que passa para o campo FTextMessage o texto de alerta. Olhando novamente a **Figura 5** também temos a classe NotEmptyStringAttribute. Este será nosso primeiro validador, sua tarefa é verificar se uma string é vazia, veja **Listagem 12**.

Listagem 11. Classe ValidationAttribute

```
ValidationAttribute = class(TCustomAttribute)
  private
    FTextMessage: string;
  public
    constructor Create(const TextMessage: string);
    function IsValid(item: TValue): boolean; virtual; abstract;
    property TextMessage : string read FTextMessage write
     FTextMessage;
  end;

constructor ValidationAttribute.Create(const TextMessage:
string);
begin
  FTextMessage := TextMessage;
end;
```

Listagem 12. Validador NotEmptyStringAttribute

```
  NotEmptyStringAttribute = class(ValidationAttribute)
  public
    function IsValid(item: TValue): boolean; override;
  end;

function NotEmptyStringAttribute.IsValid(item: TValue): boolean;
begin
  result := (item.AsString <> '');
end;
```

Observe a implementação do método IsValid. Caso o item não esteja vazio é retornado true, indicando que o valor passou no teste. Mas veja, item é do tipo TValue, o que é isso? Vamos explicar. TValue é um tipo Record, definido na unit Rtti.pas. É basicamente utilizado para passagem de valores pois nele é possível armazenar valor e informação de reflexão de uma instância qualquer. Se abrimos a unit Rtti.pas e vermos a implementação de TValue, vamos encontrar vários métodos para obtenção do valor contido em um TValue, como mostra **Listagem 13.**

Listagem 13. Métodos de TValue

```
function AsString: string;
function AsInteger: Integer;
function AsExtended: Extended;
function AsInt64: Int64;
function AsObject: TObject;
function AsClass: TClass;
function AsBoolean: Boolean;
```

Continuando com as outras classes de nosso framework, temos MinLenghtStringAttribute. Sua principal função é validar se uma determinada propriedade de um controle da VCL ou objeto possui a quantidade mínima de caracteres definidos. Vamos ver seu código na **Listagem 14.**

A primeira coisa a se notar nesse validador é a existência de dois construtores. O primeiro solicita apenas a quantidade mínima de caracteres e a mensagem de alerta, o segundo por sua vez solicita um parâmetro a mais, o TargetProperty. Esse validador poderá ser aplicado em um objeto de negócio ou em um controle da VCL, por exemplo um TEdit. Quando o aplicarmos na VCL é preciso informar qual a propriedade do controle VCL que será testada. Isso será informado no parâmetro TargetProperty. Veja agora como o método IsValid foi implementado. Verificamos primeiramente se a propriedade

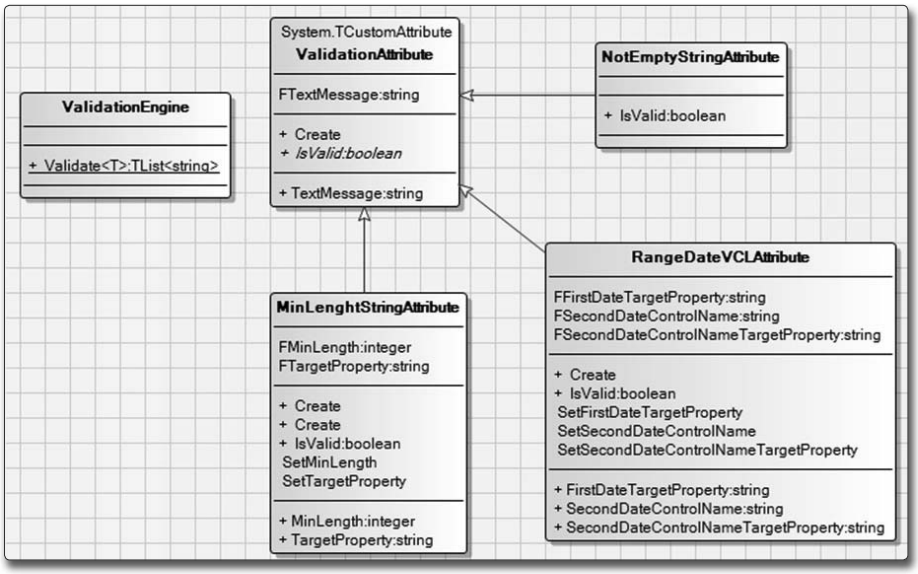


Figura 5. Classes do Validador

Listagem 14. Validador MinLenghtStringAttribute

```
MinLenghtStringAttribute = class(ValidationAttribute)
  private
    FMinLength: integer;
    FTargetProperty: string;
    procedure SetMinLength(const Value: integer);
    procedure SetTargetProperty(const Value: string);
  public
    constructor Create(const MinLength: integer; TextMessage:
     string); overload;
    constructor Create(const MinLength: integer; TargetProperty:
     string; TextMessage: string); overload;
    function IsValid(item: TValue):boolean; override;
    property MinLength: integer read FMinLength write
     SetMinLength;
    property TargetProperty: string read FTargetProperty write
      SetTargetProperty;
  end;

constructor MinLenghtStringAttribute.Create(const MinLength:
integer;
  TextMessage: string);
begin
  inherited Create(TextMessage);
  FMinLength := MinLength;
end;

constructor MinLenghtStringAttribute.Create(const MinLength:
integer;
  TargetProperty: string; TextMessage: string);
begin
  Create(MinLength,TextMessage);
  FTargetProperty := TargetProperty;
end;

function MinLenghtStringAttribute.IsValid(item: TValue): boolean;
var
 contexto : TRttiContext;
 instancia : TRttiInstanceType;
begin
  if TargetProperty <> '' then
  begin
    contexto := TRttiContext.Create;
    try
      instancia := (contexto.GetType(item.AsObject.ClassInfo) as
      TRttiInstanceType);
      result := Length(instancia.GetProperty(TargetProperty).
      GetValue(item.AsObject).AsString) >= FMinLength;
    finally
      contexto.Free;
    end;
  end
  else
    result := Length(item.AsString) >= FMinLength;
end;

procedure MinLenghtStringAttribute.SetMinLength(const Value:
integer);
begin
  FMinLength := Value;
end;

procedure MinLenghtStringAttribute.SetTargetProperty(const Value:
string);
begin
  FTargetProperty := Value;
end;
```

TargetProperty foi preenchida, se sim, indica que o atributo foi aplicado sobre um controle VCL e portanto, temos que usar reflexão para obtermos seu valor. Para isso, fizemos como nos exemplos anteriores. Criamos um contexto de reflexão e extraímos as informações de Item. Contudo veja que estamos fazendo um Cast para o tipo TRttiInstanceType. Esse tipo representa a instância do objeto junto com as informações de sua classe. Dessa forma é possível obter o valor do componente, como é mostrado a seguir:

```
instancia.GetProperty(TargetProperty).GetValue(item.AsObject).AsString
```

Podemos utilizar o método GetProperty e passar a ele o nome da propriedade do objeto item que precisamos acessar. Esse método retorna um TRttiProperty que por sua vez possui o método GetValue, utilizado para recuperar o conteúdo da propriedade baseado na instância passada como parâmetro. Nesse caso, se nosso validador estiver posicionado sobre um TEdit, item será sua instância. De posse do valor da propriedade, então apenas verificamos se ela possui a quantidade mínima de caracteres. Mas se o validador estiver aplicado em um objeto comum, de negócio, apenas convertemos Item para string e verificamos seu tamanho, pois não é necessário reflexão nesse caso. Vamos entender o porquê.

Nossos validadores são na verdade classes Custom Attribute que poderão, quando previsto, ser aplicadas em objetos e controles VCL. O uso do MinLenghtStringAttribute para objeto e controle VCL seria respectivamente:

```
[MinLenghtStringAttribute(5, 'Quantidade mínima para busca é 3
caracteres')]
Property TextoBusca: string read FTextoBusca write FTextoBusca;


[MinLenghtStringAttribute(5,'Text','Quantidade mínima para Edit1 é 5
caracteres')]
Edit1: TEdit;
```

No primeiro caso não aplicamos reflexão porque o atributo está sobre uma propriedade string, ela será o Item utilizado em IsValid. Mas no segundo caso temos um TEdit que também será o Item de IsValid. Por isso é preciso informar qual propriedade dele que será testada.

Chegamos então ao último validador, RangeDateVCLAttribute. Ele foi desenvolvido para fazer aquela verificação, "chata", de validação de período de datas e será estritamente utilizado em controles da VCL, por isso incluí em seu nome o sufixo VCL. Na **Listagem 15** vemos sua definição e implementação.

Nele temos apenas um construtor definido que recebe como parâmetro o nome da propriedade que contém a primeira data, o nome do controle que contém a segunda data, a propriedade do controle que contém a segunda e a mensagem de alerta. Se vamos comparar um período é preciso compará-lo com um segundo valor, por isso indicamos no construtor qual controle possui a segunda data desse período. Novamente o método que merece destaque é o IsValid. Separamos a validação em dois passos. Primeiro é preciso obter a primeira data para depois obter a segunda para então compará-las. O primeiro passo é feito no contexto contextoFirstDate. Nele extraímos o conteúdo da propriedade indicada, como já foi feito anteriormente. Para obtermos o segundo valor é preciso fazer algo diferente. Como foi passado o nome do controle e a propriedade, é necessário encontrar esse controle no Parent do controle que contém a primeira data. Digamos que você coloque em um formulário dois controles TEdit's. O parent de cada um é o formulário onde estão colocados. Um objeto que pode conter outros (TForm, TPanel, etc) mantém um lista com os objetos contidos nele. Assim podemos localizar o segundo controle e usar reflexão para extrair seu valor, como é mostrado no trecho de código da **Listagem 16**.

Listagem 15. Validador RangeDateVCLAttribute

```
  RangeDateVCLAttribute = class(ValidationAttribute)
  private
    FFirstDateTargetProperty: string;
    FSecondDateControlName: string;
    FSecondDateControlNameTargetProperty: string;
    procedure SetFirstDateTargetProperty(const Value: string);
    procedure SetSecondDateControlName(const Value: string);
    procedure SetSecondDateControlNameTargetProperty(const Value:
    string);
   public
    constructor Create(const FirstDateTargetProperty: string;
     SecondDateControlName: string; SecondDateControl
     NameTargetProperty: string; TextMessage: string);overload;
    function IsValid(item: TValue):boolean; override;
    property FirstDateTargetProperty: string read
     FFirstDateTargetProperty write SetFirstDateTargetProperty;
    property SecondDateControlName:string read
     FSecondDateControlName write SetSecondDateControlName;
    property SecondDateControlNameTargetProperty:
     string read FSecondDateControlNameTargetProperty write
      SetSecondDateControlNameTargetProperty;
   end;

constructor RangeDateVCLAttribute.Create(const
FirstDateTargetProperty: string;
  SecondDateControlName, SecondDateControlNameTargetProperty,
  TextMessage: string);
begin
  inherited Create(TextMessage);
  FFirstDateTargetProperty := FirstDateTargetProperty;
  FSecondDateControlName := SecondDateControlName;
  FSecondDateControlNameTargetProperty :=
  SecondDateControlNameTargetProperty;
end;

function RangeDateVCLAttribute.IsValid(item: TValue): boolean;
var
 contextoFirstDate : TRttiContext;
 instanciaFirstDate : TRttiInstanceType;

 contextoSecondDate : TRttiContext;
 instanciaSecondDate : TRttiInstanceType;

 FirstDate, SecondDate: string;
 Parent: TWinControl;
 obj: TControl;
 x: integer;
begin
  contextoFirstDate := TRttiContext.Create;
  try
    instanciaFirstDate := (contextoFirstDate.GetType
    (item.AsObject.ClassInfo) as TRttiInstanceType);
    FirstDate := instanciaFirstDate.GetProperty(FirstDateTarget
    Property).GetValue(item.AsObject).AsString;
    Parent := TWinControl(instanciaFirstDate.
     GetProperty('Parent').GetValue(item.AsObject).AsObject);
  finally
    contextoFirstDate.Free;
  end;

  for x:=1 to Parent.ControlCount-1 do
  begin
    obj := Parent.Controls[x];
    if obj.Name = SecondDateControlName then
    begin
      contextoSecondDate := TRttiContext.Create;
      try
        instanciaSecondDate := (contextoSecondDate.GetType
         (obj.ClassInfo) as TRttiInstanceType);
        SecondDate := instanciaSecondDate.GetProperty(SecondDate
         ControlNameTargetProperty).GetValue(obj).AsString;
      finally
        contextoSecondDate.Free;
      end;
      break;
    end;
  end;

  try
    result := StrToDate(FirstDate) <= StrToDate(SecondDate);
  except
   result := false;
  end;

end;

procedure RangeDateVCLAttribute.SetFirstDateTargetProperty(const
Value: string);
begin
  FFirstDateTargetProperty := Value;
end;

procedure RangeDateVCLAttribute.SetSecondDateControlName(const
Value: string);
begin
  FSecondDateControlName := Value;
end;

procedure RangeDateVCLAttribute.
SetSecondDateControlNameTargetProperty(
  const Value: string);
begin
  FSecondDateControlNameTargetProperty := Value;
end;
```

Nós obtemos o Parent do controle que contém a primeira data para então percorrer sua lista de controles e localizar, pelo nome, aquele que contém a segunda data. Após achá-lo aplicamos a RTTI para extrair o valor da propriedade passada como parâmetro. E por fim, comparamos as datas:

```
try
  result := StrToDate(FirstDate) <= StrToDate(SecondDate);
except
  result := false;
end;
```

Listagem 16. Localizando controles, usando parent

```
Parent := TWinControl(instanciaFirstDate.GetProperty('Parent').
GetValue(item.AsObject).AsObject);

for x:=1 to Parent.ControlCount-1 do
  begin
    obj := Parent.Controls[x];
    if obj.Name = SecondDateControlName then
    begin
      contextoSecondDate := TRttiContext.Create;
      try
        instanciaSecondDate := (contextoSecondDate.GetType(obj.
         ClassInfo) as TRttiInstanceType);
        SecondDate := instanciaSecondDate.GetProperty
        (SecondDateControlNameTargetProperty).GetValue(obj).
         AsString;finally
        contextoSecondDate.Free;
      end;
      break;
    end;
  end;
```

Até o momento apenas criamos as regras, os validadores que poderão ser aplicados. Mas é preciso criar a classe que irá executar esses validadores. Isso é feito em ValidationEngine. Ela possui um único método estático, Validate<T>, como mostra **Listagem 17**.

O método Validate é o que coloca tudo para funcionar, fazendo uso da RTTI e de Generics. Nele criamos o contexto de reflexão para extrair as propriedades e campos do tipo em T. Obtemos também a instância do objeto passado como parâmetro. Quando obtemos a lista de propriedades usando o método GetProperties, fazemos um laço "for in" para recuperar os CustomAttributes que possam existir em cada propriedade utilizando o método GetAttributes. Se a propriedade possuir um CustomAttribute testamos seu tipo pelo operador is. Se o CustomAttribute existente for um dos que desenvolvemos, executamos seu método IsValid e caso seja retornado false, armazenamos em uma lista genérica a mensagem de erro que foi definida. Todo esse trabalho de reflexão é realizado para propriedades e campos. Por fim, o método retorna a lista contendo as mensagens de erro.

Utilizando o framework

Chegou a hora de por tudo em prática. Vamos utilizar o projeto que foi criado para o framework, como ele já possui um formulário, vamos deixá-lo como mostra a **Figura 6.**

Listagem 17. ValidationEngine

```
  ValidationEngine = class
  public
    class function Validate<T>(item: TObject): TList<string>;
  end;

class function ValidationEngine.Validate<T>(item: TObject):
TList<string>;
var
 contexto : TRttiContext;
 tipo : TRttiType;
 propriedade : TRttiProperty;
 campo: TRttiField;
 atributo : TCustomAttribute;
 instancia : TRttiInstanceType;

 ErrorList: TList<string>;
begin
  ErrorList := TList<String>.Create;
  contexto := TRttiContext.Create;
  try
    tipo := contexto.GetType(typeinfo(T));
    instancia := (contexto.GetType(item.ClassInfo) as
     TRttiInstanceType);
    for propriedade in tipo.GetProperties do
      for atributo in propriedade.GetAttributes do
      begin
        if atributo is NotEmptyStringAttribute then
        begin
          if not NotEmptyStringAttribute(atributo).
          IsValid(instancia.GetProperty(propriedade.Name).
          GetValue(item)) then
          ErrorList.Add(NotEmptyStringAttribute(atributo).
          TextMessage);
        end;

        if atributo is MinLenghtStringAttribute then
        begin
          if not MinLenghtStringAttribute(atributo).
          IsValid(instancia.GetProperty(propriedade.Name).
          GetValue(item)) then
          ErrorList.Add(MinLenghtStringAttribute(atributo).
          TextMessage);
        end;

        if atributo is RangeDateVCLAttribute then
        begin
          if not RangeDateVCLAttribute(atributo).
          IsValid(instancia.GetProperty(propriedade.Name).
          GetValue(item)) then
          ErrorList.Add(RangeDateVCLAttribute(atributo).
          TextMessage);
        end;
      end;

    for campo in tipo.GetFields do
    begin
      for atributo in campo.GetAttributes do
      begin
        if atributo is  NotEmptyStringAttribute then
        begin
          if not NotEmptyStringAttribute(atributo).
          IsValid(instancia.GetField(campo.Name).GetValue(item))
          then
          ErrorList.Add(NotEmptyStringAttribute(atributo).
          TextMessage);
        end;

        if atributo is MinLenghtStringAttribute then
        begin
          if not MinLenghtStringAttribute(atributo).
          IsValid(instancia.GetField(campo.Name).GetValue(item))
          then
          ErrorList.Add(MinLenghtStringAttribute(atributo).
          TextMessage);
        end;

        if atributo is RangeDateVCLAttribute then
        begin
          if not RangeDateVCLAttribute(atributo).
          IsValid(instancia.GetField(campo.Name).GetValue(item))
          then
          ErrorList.Add(RangeDateVCLAttribute(atributo).
          TextMessage);
        end;
      end;
    end;

  finally
    contexto.Free;
  end;
  result := ErrorList;

end;
```

Figura 6. Formulário para teste

Temos nesse formulário um TEdit, dois TMaskedEdit's que irão representar um período de datas, um botão e um ListBox que será utilizado para conter as mensagens de erro. Na seção uses inclua a unit que contém o framework, a Validation.pas. Com isso podemos decorar os controles com as validações. Deixe a estrutura da classe do formulário como mostra a **Listagem 18.**

Listagem 18. Aplicando os validadores

```
TForm1 = class(TForm)
    Button1: TButton;

    [MinLenghtStringAttribute(5,'Text','Quantidade mínima para
    Edit1 é 5 caracteres')]
    Edit1: TEdit;

    Label1: TLabel;

    [RangeDateVCLAttribute('Text','MaskEdit2','Text','Período de
    datas inválido')]
    MaskEdit1: TMaskEdit;

    MaskEdit2: TMaskEdit;
    Label2: TLabel;
    ListBox1: TListBox;
 private
    FNome: string;
    procedure SetNome(const Value: string);
    { Private declarations }
 public
    { Public declarations }
    [NotEmptyStringAttribute('A propriedade nome não pode ser
     vazia')]
    property Nome: string read FNome write SetNome;
 end;
```

Veja que adicionamos um atributo em Edit1. Nele informamos a quantidade mínima de caracteres, a propriedade que vamos testar e a mensagem de erro. Já em MaskEdit1 passamos a propriedade que contém a primeira data, o nome do controle que possui a segunda data, em qual propriedade ela está armazenada e qual a mensagem de erro. Definimos também uma propriedade Nome para mostrar o uso do primeiro validador em objetos de negócio. Nele aplicamos o atributo NotEmptyStringAttribute, que exige apenas a mensagem de erro. Mas para validar é preciso utilizar a classe ValidationEngine.

Vamos criar um método chamado Testar e implementá-lo como mostra a **Listagem 19**.

Listagem 19. Executando a validação

```
procedure TForm1.Testar;
var
 Lista: TList<string>;
 s: String;
 texto:string;
begin
 Lista := ValidationEngine.Validate<TForm1>(self);
 Listbox1.Items.BeginUpdate;
 ListBox1.Items.Clear;
 for s in Lista do
   ListBox1.Items.Append(s);
   ListBox1.Items.EndUpdate;
end;
```

Veja que temos uma variável do tipo TList<string> que terá os resultados do método ValidationEngine.Validate. Observe em Validate passamos o tipo do formulário, que é sua classe e depois Self, que é sua instância. Em seguida apenas adicionamos no ListBox1 as strings que contêm as mensagens de erro. Agora no evento OnClick do botão chame o método Testar. Execute a aplicação e veja o resultado. Se você informar um período inválido, como estamos executando dentro da IDE veremos uma exceção caso os dois TMaskEdit não estejam preenchidos. Se a aplicação for executada fora da IDE não veremos a exceção. Para deixar mais dinâmica a validação dos dados, podemos no FormShow do formulário executar o método Testar e também no método OnChange do Edit1 e dos TMaskEdit, assim, a validação ocorre conforme a digitação acontece, como mostra a **Figura 7**.

Figura 7. Demonstração em execução

Conclusão

Os recursos que foram melhorados/adicionados no Delphi 2010 nos permitem criar uma boa arquitetura para nossos sistemas. Podemos centralizar, generalizar e reutilizar coisas que são comuns. A exemplo disso temos aqui um mini framework de validação. Em todo projeto é necessário se validar algo, e ter isso de uma forma que possa ser reutilizável é ótimo, ainda mais se for fácil de ser utilizada. Aplicar um atributo em um controle ou em uma propriedade é muito simples e exige um esforço mínimo.

Outra ideia é geração de arquivos textos. Porque não criar um framework de geração de arquivos, onde através de atributos aplicados em classes e propriedades sejam definidos o tamanho, posição, caractere delimitador etc. Fica aqui a ideia. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Links

Validation Application Block
http://msdn.microsoft.com/en-us/library/cc309509.aspx

Utilizando VAB em aplicações ASP.NET
http://www.15seconds.com/Issue/070607.htm

Generics, Facilitando a vida do desenvolvedor
http://msdn.microsoft.com/pt-br/library/cc580580.aspx

Usando Custom Attributes
http://edn.embarcadero.com/article/30250

Para Delphi 2009/2010

Do premiado autor da série
Dominando o Delphi: a Bíblia

Domine todos os novos recursos do Delphi, como o suporte a Unicode, os tipos genéricos, os métodos anônimos, os controles Ribbon (e outros novos controles), o DBExpress 4, o novo DataSnap e as novas funcionalidades do IDE.

Os novos recursos do Delphi são abordados em profundidade, com inúmeros exemplos de como são usados na prática. Os problemas que podem ser encontrados ao atualizar programas escritos em versões anteriores do Delphi (no Delphi 7 ou Delphi 2007, por exemplo), em especial devido à introdução do suporte a Unicode, são discutidos em detalhes.

Recursos Avançados do Delphi é indispensável para todos os desenvolvedores, mesmo iniciantes, já que apresenta informações que não são encontradas em outros livros, e que são necessárias para escrever programas que façam proveito pleno dos novos recursos do Windows XP, Vista e 7.

416 páginas – lançamento em 09/2009 – R$ 79,00

Nas melhores livrarias ou em
www.infopress.com.br

Nesta seção você encontra artigos para iniciantes na linguagem Delphi

Introdução à POO – Parte 6

Aprenda de uma forma simples a programar de forma correta

Guinther Pauli
guinther.pauli@gmail.com
Blog: http://guintherpauli.blogspot.com
http://twitter.com/guintherpauli
Tem 22 anos de experiência na área de desenvolvimento e arquitetura de software. Certificado em Delphi 3, 5, 6, 7, 2005, 2006, Delphi para Web e Kylix, certificado Microsoft MCP, MCAD, MCSD.NET, MCTS e MCPD (C#, ASP.NET). Editor da Revista ClubeDelphi e da Revista .NET Magazine Brasil, professor de Pós-Graduação da UNIVEM-SP. Especialista em desenvolvimento Delphi, C#, .NET, ASP.NET, atuando também em projetos de migração orientando empresas e desenvolvedores na adotação de melhores práticas. Foi arquiteto e um dos desenvolvedores, voluntário, do site de doações on-line do projeto Criança Esperança da Rede Globo / Unesco.

Resumo do DevMan

Este artigo mostra como usar a orientação a objetos e todos os fundamentos vistos até aqui em um cenário real e crítico, um sistema de vestibulares e correção de provas.

Do que trata o artigo
Este artigo aborda a orientação a objetos com o Delphi, usando uma metodologia simples, didática, de fácil aprendizado. Veremos na teoria, e também na prática, todos os conceitos, fundamentos e recursos oferecidos pelo Delphi para promover a POO. Nesta parte final, um exemplo real.

Para que serve
A POO pode e deve ser aplicada de forma inteligente, ela serve para construir sistemas mais robustos, mais confiáveis, de fácil manutenção, que permitam maior reaproveitamento de código.

Em que situação o tema é útil
A POO é útil em qualquer sistema, seja ele Web, Desktop, Mobile, não importa o tipo de aplicação. Os conceitos apresentados até aqui foram aplicados em um exemplo simples, agora serão utilizados em uma situação real.

Chegamos à parte final do nosso curso de Delphi e OO. Aprendemos muito até aqui, os pilares da programação orientada a objetos foram todos estudados: herança, polimorfismo, abstração e encapsulamento. Através da elaboração de um pequeno framework, estudamos como a Delphi Language promove a OO. Criamos classes, propriedades, métodos estáticos, abstratos e virtuais, especificadores de visibilidade / modificadores, estudamos construtores, eventos, relacionamentos entre classes como herança e associação, polimorfismo, interfaces e mais. Estendemos nosso framework comprovando que ele é flexível e expansível.

Muito bem. Vimos que a OO funcionou legal com carrinhos e pessoas, afinal, usamos um exemplo bem simples (até porque estamos na sessão Easy). Lembrando ainda que a maioria dos exemplos em Delphi e OO se baseiam em situações não reais (como o que vimos até aqui). Então, como colocar em uma aplicação real tudo o que aprendemos até aqui?

Entender o polimorfismo para quem é novo na OO é difícil, sair da teoria e colocá-lo em prática é muito difícil. Entendê-lo e colocá-lo em prática em um exemplo real, quase impossível. Esse é um grande mito que vamos derrubar hoje.

Cenário, jogo e regras

Há vários anos, mais exatamente em 2000, trabalhei como desenvolvedor em uma grande universidade aqui da minha região. Um dos principais sistemas que desenvolvemos foi o que controlava todos os concursos vestibulares da instituição, incluindo a parte mais crítica: correção e listão. Já adianto que por motivos óbvios não é possível mostrar qualquer código de implementação aqui, apenas vou prototipar uma solução semelhante a que defini na época.

Acredito que todos aqui já prestaram vestibular alguma vez na vida. Todos sabem que não é fácil ter o nome no listão. Isso eu também sabia. O que eu não sabia é que, quando estava do outro lado (como programador), colocar um nome no listão não era nada simples. Se existe um software que não permite qualquer tipo de falha, esse é um sistema de vestibulares. Não seria nada interessante enviar um novo listão para vários jornais e sites dois dias após a divulgação oficial, informando que o mesmo foi anulado devido a uma "falha no sistema".

Enfim, tínhamos uma missão crítica e de muita responsabilidade a resolver: criar um sistema para controle de processos vestibulares. Não vou me deter aqui em explicar todas as regras de um processo de seleção, acho que a maioria sabe, até porque nosso objetivo aqui não é criar um sistema, mas examinar como a OO resolve problemas de forma muito elegante. Quem me dera que fosse simplesmente calcular uma nota e fazer um select para ordenar pela nota capturando do 1 até n, onde n é o número de vagas do curso.

Mas basicamente podemos considerar o seguinte: existiam três tipos de leituras que eram feitas durante o vestibular, a leitura dos cartões de inscrição (naquela época inscrição pela internet não existia), correção das provas propriamente ditas e correção da nota da redação. Uma máquina se encarregava de ler os cartões respostas que gerava então um arquivo TXT (normal), que devia a seguir ser processado pelo software. Obviamente, cada tipo de leitura tinha suas particularidades. Uma ficha de inscrição contém dados do vestibulando, como nome, data de nascimento, opção de curso, língua estrangeira. A resposta de uma prova contém o número de inscrição e as letras marcadas pelo candidato. Mas as leituras possuem muitas, mas muitas coisas em comum. Sem dúvida estava diante de um belo cenário para colocar em prática os ensinamentos do meu professor Marco Cantu, ensinados em suas "bíblias".

A solução

A **Listagem 1** mostra um protótipo do que seria uma possível solução para o problema. Temos uma classe base chamada TCorrecao, que contém todas as funcionalidades básicas de um processamento de leitura. Por exemplo, todas

Listagem 1. Classes "core" do processo de correção

```
// Esta unit é um protótipo e não contém implementação
unit uCorrecao;

interface

uses Classes, SysUtils;

type
  ECorrecaoException = class (Exception);

  TCorrecao = class (TObject)
  private
    FArquivo: TStrings;
    FLog: TStrings;
    procedure VerificarIntegridadeArquivo();
  protected
    procedure CarregarInscricoes(); virtual;
    function Merge(Original,Nova : string) : string;
    procedure LimparAnteriores(); virtual; abstract;
  public
    constructor Create;
    destructor Destroy; override;
    procedure SetArquivo(Arquivo: TStrings);
    procedure CarregarArquivoLeitora();
    procedure VerificarPreRequisitos(); virtual;
    procedure ExtrairMelhoresRespostas();
    procedure Carregar(); virtual; abstract;
    procedure Corrigir(); virtual; abstract;
    procedure EnviarDadosParaBanco(); virtual; abstract;
    procedure Finalizar(); virtual;
    procedure AddLog(Mensagem : string);
    function GetLog: string;
  end;

  TCorrecaoInscricao = class(TCorrecao)
  private
    procedure CarregarLinguas();
    procedure CarregarCursos();
  public
    procedure VerificarPreRequisitos(); override;
    procedure Carregar(); override;
    procedure Corrigir(); override;
    procedure EnviarDadosParaBanco(); override;
    procedure Finalizar(); override;
  end;

  TCorrecaoRedacao = class(TCorrecao)
  private
    procedure CalcularNotas();
    procedure Classificar();
  protected
    procedure CarregarInscricoes(); override;
  public
    procedure Carregar(); override;
    procedure Corrigir(); override;
    procedure EnviarDadosParaBanco(); override;
    procedure Finalizar(); override;
  end;

  TCorrecaoProva = class(TCorrecao)
  private
    procedure CarregarInscricoesLingua();
    procedure CarregarGabaritos();
    procedure CalcularPontoDeCorte();
    procedure CalcularNotas();
    procedure SelecionarRedacoes();
    procedure GerarDesempenhos();
    procedure Classificar();
  protected
    procedure CarregarInscricoes(); override;
  public
    procedure Carregar(); override;
    procedure Corrigir(); override;
    procedure EnviarDadosParaBanco(); override;
    procedure Finalizar(); override;
  end;

  TCorrecaoTipo = (Inscricao, Redacao, Prova);

  TCorrecaoFactory = class(TObject)
    class function CreateCorrecao(Tipo: TCorrecaoTipo):
    TCorrecao;
  end;

implementation

{ TCorrecao }
// Aperte Shift+Ctrl+C para gerar os cabeçalhos dos métodos das
classes
// A implementação das correções não é feita neste exemplo

{ TCorrecaoFactory }

class function TCorrecaoFactory.CreateCorrecao(Tipo:
TCorrecaoTipo): TCorrecao;
begin
  case Tipo of
    Inscricao : Result := TCorrecaoInscricao.Create();
    Redacao : Result := TCorrecaoRedacao.Create();
    Prova : Result := TCorrecaoProva.Create();
  end;
end;
```

as correções devem carregar um arquivo texto. Todas as correções têm pré-requisitos: para corrigir uma prova, o candidato deve obviamente estar inscrito. Para corrigir uma redação, ele deve ter sido selecionado no ponto de corte. Todas as leituras têm melhores respostas, isso porque uma mesma ficha de um candidato pode ser passada por uma leitora de cartões inúmeras vezes, com sensibilidades diferentes, até que seja obtida a melhor resposta. Por exemplo, a leitura para um candidato de inscrição 1028 poderia vir "1028AB*D". Na segunda, "1028ABCD". Ou seja, na segunda vez temos uma leitura mais consistente e devemos fazer o "merge" de ambas.

Os dados após processados precisam ser enviados ao banco de dados, em uma transação atômica, sem chance a erros nem dados inconsistentes. Se a correção for processada novamente, todos os dados anteriores devem ser apagados. Além disso, uma correção precisa, obviamente, ser corrigida. Nesse caso, respostas são comparadas a gabaritos. Em uma inscrição, as opções de língua estrangeira devem ser carregadas antes. Como você pode notar, existem coisas comuns a todas as classes, ou pelo menos, **têm o mesmo nome, mas com implementações diferentes**. Polimorfismo. Como você pode ver na mesma listagem, as classes descendentes implementam métodos virtuais definidos na classe base. Alguns métodos são totalmente abstratos, eles só estão lá para poderem ser chamados. Nas implementações, feita por cada uma das classes, esses métodos virtuais chamam outros métodos internos (privados) à classe concreta específica. Por exemplo, o Carregar() definido em TCorrecao e sobrescrito em TCorrecaoInscricao vai ser implementado chamando dois métodos privados para carregar línguas estrangeiras e cursos.

E onde entra o banco de dados? Bom, vale ressaltar que naquela época não estávamos muito bem no que diz respeito a acesso a dados. Tínhamos apenas BDE. O banco de dados era o DB2, da IBM, muito bom por sinal. Mas já tínhamos o ClientDataSet! Lembro-me até hoje que a primeira versão da correção levou 4 horas para processar cerca de 4 mil provas. A versão final, usando ClientDataSet's, levou 3 minutos. Qual o segredo? Tudo era feito em memória: correção, comparação com gabaritos, extração de respostas, merge, cálculo de notas, classificação etc. Quando todos os dados estavam computados em memória, em uma única transação atômica pegávamos os dados dos ClientDataSet's e jogávamos no banco.

Costumo dizer que todas as regras básicas da OO e Patterns se resumem a apenas duas: **separar** e **esconder**. Então, assim como meu form que fazia a interface com o usuário não sabia nada a respeito de correções, as correções também não sabiam muito sobre o BD. Todo o acesso e SQL's estavam concentrados em DataModule's (a SQL do listão tinha mais de 500 linhas). Não existe acesso a dados em uma classe de negócio, isso é papel do DM. Outro detalhe que você deve ter reparado, notou que os métodos praticamente não têm parâmetros em suas assinaturas? Isso é esconder. Lembre-se, mudar uma interface, uma regra, no meio do jogo, pode causar um impacto bem incômodo (interfaces são imutáveis). Então, nada melhor do que deixar tudo escondido entre os begin's e end's.

No final da listagem podemos encontrar uma implementação bem simples e rudimentar, mas bastante útil, de um padrão de projeto chamado Factory. Basicamente é uma classe que recebe um tipo de correção (enum) e devolve uma classe concreta a quem pediu, o form.

O Assistente

Confesso, a versão beta do sistema era complexa para o usuário. Ele deveria clicar em várias opções de menu para processar uma correção. Primeiro, deveria carregar um arquivo. Depois extrair as respostas. Depois corrigir. Usuários não sabem nem devem saber essas regras muito menos a ordem em que devem ser feitas. O release final reduziu drasticamente esta complexidade. Tudo o que o usuário deveria fazer, presumindo que todos os pré-requisitos estavam cadastrados (como cursos, línguas etc.) era clicar em um botão em um assistente e aguardar o processo ser feito. A **Figura 1** mostra o assistente. A **Listagem 2** mostra uma ideia de como seria o código deste form.

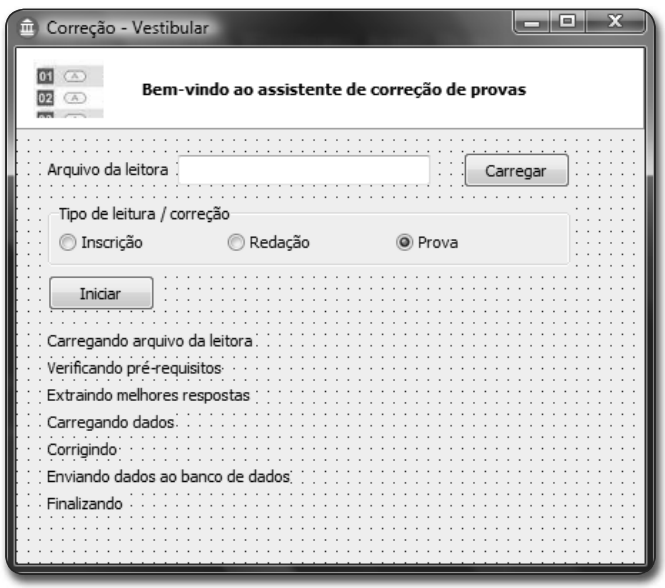


Figura 1. Assistente para processar correções

Veja algumas coisas muito interessantes aqui. O formulário não tem a menor ideia de como uma correção vai ser feita, apesar do usuário poder escolher o **tipo**. O que fizemos foi declarar um objeto da classe **base**, TCorrecao, e dentro dele colocar uma implementação concreta, obtida a partir da Factory. A partir daí, o form começa a disparar os métodos polimórficos e a OO cuida do resto. Note, **não existe IF's. Nem RTTI**. É tudo polimorfismo e abstração. Para dar um feedback elegante ao usuário, defini um método que recebe como parâmetro um "procedure of object" (ou delegate, para quem já desenvolve em .NET). Este procedimento chama então esse ponteiro, que

nada mais é que o método polimórfico da classe base e destaca uma label que indica em que etapa do processamento estamos. Tudo isso protegido com try finally.

Veja que toda exceção gerada nas classes de negócio (correções) vai parar na UI. Jamais damos um feedback ao usuário a partir de uma classe de negócio. Deixamos ela chegar no form e a tratamos. Nesse caso, colocamos em um LOG, que a propósito, é comum a todas as classes de correção. A correção também define um tipo específico de Exception. Isso é ótimo para sabermos, por exemplo, quando uma exceção acontece na correção, se ela foi gerada devido a uma falha no banco, no arquivo etc. ou se ela foi levantada pela própria classe de correção (ou seja, por mim, com raise). Se ao processar uma inscrição eu verificar que faltou um pré-requisito (as línguas estrangeiras não estão cadastradas), levanto uma ECorrecaoException que vai parar no form e dar esse feedback ao usuário (e mais, já perguntar se ele quer abrir o form para cadastrar as línguas já que não o fez). A **Figura 2** mostra o demo em execução, dando um feedback ao usuário. Note que simulei o processamento dando um sleep no código.

Figura 2. Assistente chamando métodos polimórficos e dando feedback

Conclusão

Como comprovamos aqui, a OO não precisa ser usada com complexidade. Aliás, ela nos ajuda a resolver problemas críticos e complexos, como um sistema de vestibular. O que vimos aqui hoje é algo extremamente simples, mas

Listagem 2. Código da unit do assistente

```
unit uFrmCorrecao;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms,
  Dialogs, WideStrings, StdCtrls, jpeg, ExtCtrls, uCorrecao;

type

  TAcao = procedure of object;

  TFrmCorrecao = class(TForm)
    Panel1: TPanel;
    Label1: TLabel;
    Button1: TButton;
    Image1: TImage;
    Edit1: TEdit;
    Label2: TLabel;
    Button2: TButton;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    RadioGroup1: TRadioGroup;
    Label9: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
    procedure Execute(ALabel: TLabel; Correcao: TCorrecao; Acao:
    TAcao);
  public
    { Public declarations }
  end;

var
  FrmCorrecao: TFrmCorrecao;

implementation

{$R *.dfm}

procedure TFrmCorrecao.Button1Click(Sender: TObject);
var
  Correcao: TCorrecao; ← único ponto de ligação entre form e negócio
  Tipo: TCorrecaoTipo;
begin
  Tipo := TCorrecaoTipo(RadioGroup1.ItemIndex);
  Correcao := TCorrecaoFactory.CreateCorrecao(Tipo);
  try
    try
      Execute(Label3,Correcao,Correcao.CarregarArquivoLeitora);
      Execute(Label4,Correcao,Correcao.VerificarPreRequisitos);
      Execute(Label5,Correcao,Correcao.ExtrairMelhoresRespostas);
      Execute(Label6,Correcao,Correcao.Carregar);
      Execute(Label7,Correcao,Correcao.Corrigir);
      Execute(Label8,Correcao,Correcao.EnviarDadosParaBanco);
      Execute(Label9,Correcao,Correcao.Finalizar);
    except
      // Exibe log de forma simples
      // sugestão, colocar no final do assistente
      ShowMessage(Correcao.GetLog());
    end;
  finally
    Correcao.Free();
  end;
end;

procedure TFrmCorrecao.Execute(ALabel: TLabel; Correcao:
TCorrecao; Acao: TAcao);
begin
  ALabel.font.style := [fsbold];
  Application.ProcessMessages();
  try
    Acao();
    // Simula processamento ...
    Sleep(500);
    ALabel.caption := ALabel.caption+' (OK!)';
    ALabel.font.color := clSilver;
  except
    // aqui você pode tratar se E é ECorrecaoException
    on E : Exception do
    begin
      Correcao.AddLog('Houve um erro enquanto o sistema estava ' +
      ALabel.Caption);
      Correcao.AddLog(E.Message);
      ALabel.caption := ALabel.Caption + ' (Erro!)';
      ALabel.font.color := clred;
    end;
    raise; // propaga exceção para ser tratada acima
  end;
end;

end.
```

mostra que mesmo pequenos sistemas podem usar a POO e desfrutar de todos os seus recursos. Eu certamente não imaginaria outra forma de criar um sistema deste tipo e porte (e com tamanha responsabilidade nas costas do desenvolvedor) sem usar a OO. Sem dúvida, o que mostrei aqui não é a única solução possível para o problema, nem tem uma definição / implementação perfeita, mas comprova que o Delphi permite o uso da OO sem perder seu estilo RAD de ser.

Desejo sucesso a todos em seus novos projetos, agora com muita OO. Grande abraço. ●

Links

Blog do autor

http://guintherpauli.blogspot.com

http://twitter.com/guintherpauli

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Cursos Online

Web Mobile PLUS

A Revista **WebMobile** oferece para seus assinantes uma série de Cursos Online de alto padrão de qualidade .

Conheça abaixo os cursos já disponíveis:
www.devmedia.com.br/curso/webmobile

- Curso em .NET: **Aplicação completa de orçamento Doméstico no Visual Studio 2005**
- Curso em Java: **Introdução ao desenvolvimento para celulares com J2ME**

A sua melhor opção de aprendizagem!

Assine a **WebMobile** e
Comece já seu treinamento!
www.devmedia.com.br/assine

DevMedia GROUP Mais Informações: www.devmedia.com.br/central - Tel.: 21 2220-5375 / 2220-5435

A partir de agora você pode contar com a ajuda da DevMedia em seus projetos e estudos.

Chegou o Suporte On-line DevMedia

Tira-dúvidas + Professor Virtual + Certificação

A DevMedia possui um numeroso time de autores, editores e professores que juntos produzem o material que você está acostumado a encontrar em nosso site e revistas. E são exatamente esses mesmos profissionais que estarão a sua disposição para tirar suas dúvidas e ajudá-lo em seus projetos e estudos. Através de uma plataforma 100% web o Suporte DevMedia garante sigilo absoluto, eficiência e rapidez em todas as respostas. Finalmente você terá ao seu alcance um suporte de qualidade por um preço muito acessível. Consulte nossos planos.

Mais informações:
www.devmedia.com.br/suporte
21.3382.5038

Mais um serviço DevMedia group

Hospede seus sites de forma simples, rápida e ilimitada.

Plano **Básico**

- 1 domínio principal
- 50 domínios hospedados
- Transferência **ilimitada**
- Espaço **ilimitado**
- 10 contas de e-mail
- PHP, ASP e ASP.NET
- Painel de controle Plesk
- Suporte 24hs

Tudo isso por apenas

R$ **15,90** por mês
pagamento trimestral

Assine agora mesmo e descubra por que Hospedix é a hospedagem de sites preferida entre os profissionais.

- Conheça também nossos planos profissionais com registro de domínio grátis e isenção de taxas anuais enquanto for cliente.

HOSPEDIX
HOSPEDAGEM PROFISSIONAL
www.hospedix.com.br

30 DIAS GRÁTIS

Ao assinar, digite o código de desconto exclusivo para leitores desta revista e ganhe 30 dias inteiramente grátis.

RVSTMEDIA

As promoções anunciadas não são cumulativas com outras promoções e são válidas apenas para clientes Hospedix ativos.